Modeling with TDL

Copyright preeTEC.com
March 2008

Contents

The Timing Definition Language (TDL) in a nut shell 1

What is a TDL component (= module)? 2
Logical Execution Time (LET) 3
TDL sample modules 3
TDL modeling and platform deployment 6

Guided Tour of the TDL:VisualCreator tool 9

Define module *M1* 9
Start TDL:VisualCreator in Simulink® 10
Add constants *c1* and *c2* 11
Edit sensor *s* 12
Edit actuators *a1* and *a2* 14
Edit tasks *inc* and *dec* 15
Edit modes *f11* and *f12* 23
Test module M1 49
TDL source code of module M1 51
Define module *M2* 52
Add module *M2* 53
Remove sensor 53
Import module *M1* 54
Edit actuator *a* 55
Edit task *sum* 57
Edit mode *main* 65
TDL source code of module *M2* 69
Test module *M2* 69
Synchronous actuator updates and advance calculation 71
Structured Data Types 73
Generating platform-independent code for TDL modules 75
Transparent distribution of TDL modules 75

TDL:VisualCreator 79

Differences between the demonstration and product versions 79
Set-up as stand-alone tool 80
Set-up as add-on to Simulink® 80
Preliminaries 82
Adding a TDL construct 83
Editing a TDL construct 84
Deleting a TDL construct 86
TDL program text 87
TDL constructs 87
TDL Import declaration 87
TDL Constant declaration 88
TDL Type declaration 88

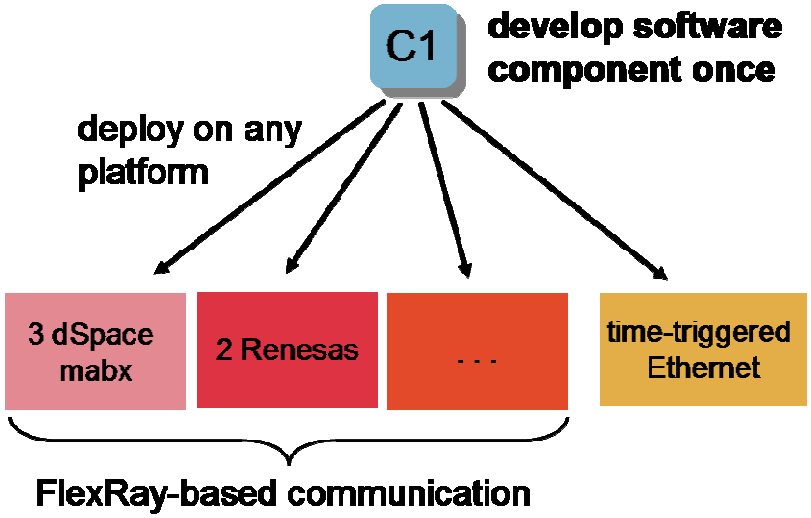
TDL Sensor declaration 90
TDL Actuator declaration 91
TDL Task declaration 92
TDL Mode declaration 94
Migration of existing Matlab/Simulink® subsystems to TDL modules 102
Functionality code generation in Simulink® 107
Simulink® Buses and TDL Structure Types 108

Appendix: The Timing Definition Language (TDL) 109

TDL delivers deterministic, portable software components 109
Determinism 109
Portability 110
Composability 110
TDL concepts 110
TDL component model 111
Comparison with the state-of-the-art 112

The Timing Definition Language (TDL) in a nut shell

The Timing Definition Language (TDL) is the first technology that allows you to develop an embedded real-time software component once and deploy it on any, potentially distributed platform that offers sufficient computing and communication resources. The following figure illustrates this portability of TDL components. The TDL component C1 could be deployed on a cluster consisting of 3 dSpace MicroAutoBox nodes, or any other platform or topology.



In other words, developers can ignore the execution platform (hardware, processors, communication protocol, operating system, and even the distribution topology) when modeling the behavior of an embedded real-time software system. A TDL program explicitly specifies the exact real-time interaction of software components with each other and the physical world. The execution platform is specified separately of the TDL program, typically after the developers came up with an appropriate TDL program that solves a particular control engineering problem. Then automatic code generators create the efficient code from the TDL program for a specific platform.

Unprecedented quality: time and value determinism. There is another important "first" regarding quality: TDL is the first technology that guarantees that the same input values at the same time instants result in the same outputs at corresponding same time instants. Furthermore, we guarantee that the behavior specified in TDL is exactly the same on any target platform. If a target platform does not offer sufficient computing and communication resources, the code generators would report that problem and no code would be generated. In other words, if code is generated, that generated code is always correct. We can offer the formal proof of that.

Significant cost reductions. The TDL benefits are significant. No other product can offer these benefits today. First, the original development costs are reduced significantly, at least by ca. 30%, in particular, if the software should be distributed. We have even measured development cost cuttings by a factor of 20 (!) and more in the context of FlexRay-based systems. The reason for the significant cost savings is that developers do not deal with platform details when specifying the functionality and timing behavior. As we guarantee that the behavior of the generated code is the same on any execution platform as specified in TDL, testing is also more straight-forward.

Another cost saving benefit is the greatly enhanced flexibility regarding the platform. For example, you could decide to move from a FlexRay-based platform to a time-triggered communication system based on Ethernet. You only need to generate code for the new platform. Even if you just change the topology of the platform by adding or removing a node (Electronic Control Unit or ECU in the automotive jargon), and thus the mapping of TDL components to nodes, you will appreciate the automatic generation of the executable code. Developers do not deal with the numerous platform details.

Simulation within Matlab/Simulink®. TDL is also seamlessly integrated in Mathworks®' Matlab/Simulink® so that the behavior of TDL components can easily be simulated. For TDL Matlab/Simulink® is yet another platform. So we also guarantee that the simulated behavior is identical with the behavior on any other platform for which code was generated.

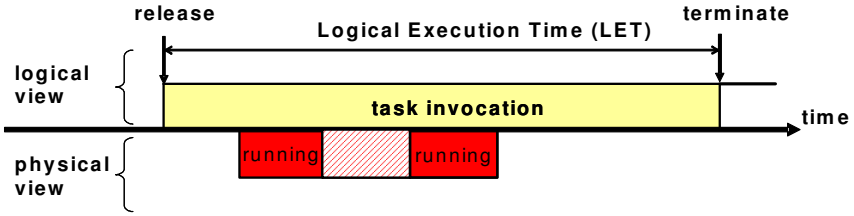
What is a TDL component (= module)?

TDL itself offers a textual notation for defining the timing behavior. TDL offers a language construct, the socalled module construct, which allows the definition of a TDL component. We use the terms module and component synonymously. A TDL module forms a unit that consists of sensors, actuators, and modes. A mode is a set of periodically executed activities. The activities are task invocations, actuator updates, and mode switches. All activities can have their own rate of execution and all activities can be executed conditionally. The tasks represent the functionality, particularly the control laws. Any suitable language can be chosen to implement the functionality. If MathWorks®' tools are used, the functionality is modeled in Matlab/Simulink®.

Actuator updates, sensor readings and mode switches are considered to be much faster than task invocations, thus they are executed in logical zero time. Whereas task invocations adhere to the so called Logical Execution Time (LET) semantics explained below.

Logical Execution Time (LET)

LET is the core abstraction of TDL that allows the abstraction from platform details and that delivers time and value determinism. LET means that the observable temporal behavior of a task is independent of its physical execution. It is only assumed that physical task execution is fast enough to fit somewhere within the logical start and end points. The following figure shows the relationship between logical and physical task execution.



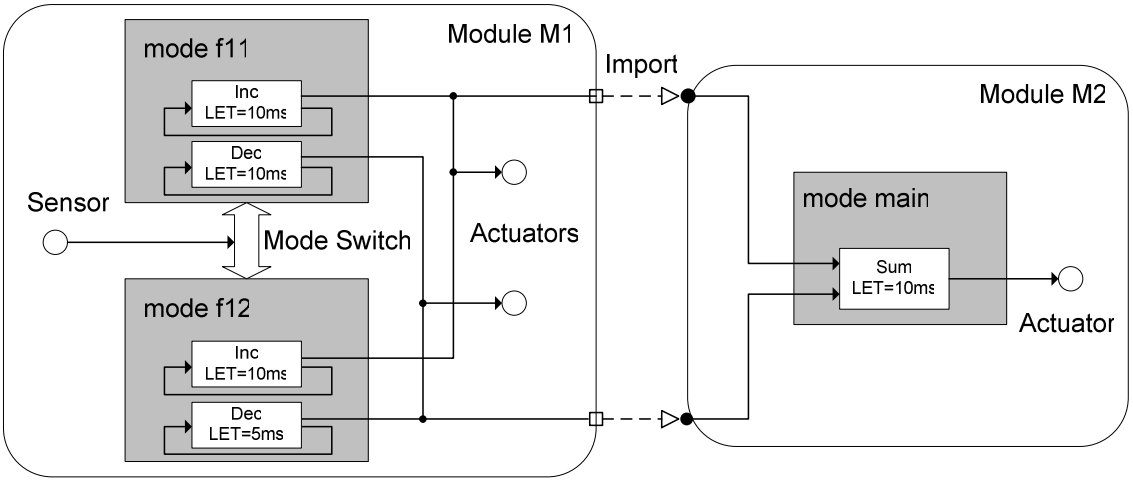
The inputs of a task are read at the release event and the newly calculated outputs are available at the terminate event. Between these, the outputs have the value of the previous execution.

LET introduces a delay for observable outputs, which might be considered a disadvantage. On the other hand, however, LET provides the cornerstone to deterministic behavior, platform abstraction as basis of portability and well-defined interaction semantics between parallel activities. It is always defined which value is in use at which time instant and there are no race conditions or priority inversions involved. If the output of a task is written to an actuator the LET can also be zero. For details, we refer to the description of the task declaration and mode declaration as well as the definition of a socalled task sequence. The following case study illustrates how TDL modules are defined and how their timing behavior is defined according to the LET abstraction.

TDL sample modules

We illustrate TDL by means of two modules with simple functionality. The figure below shows a schematic representation of the modules with their modes, tasks, and ports. Module M1 has one sensor input, two tasks called inc and dec, and two actuators connected to the output ports of the tasks. The functionality of the inc task is that it increments its output value by 10, starting with the initial value 50 up to the upper limit 200. The functionality of the dec task is that it decrements its output value by 10, starting with the initial value 200 down to the lower limit 50. The sensor is only used for switching between the two modes of the module. In mode f11 both tasks have the same LET, namely 10 ms. In mode f12 the task dec has a LET of 5 ms—it produces the output values twice as fast as task inc.

Module M2 imports module M1 and thus has access to the output ports of M1's tasks inc and dec. Module M2's task sum simply adds the outputs of M1's inc and dec tasks. The LET of task sum is 10 ms.

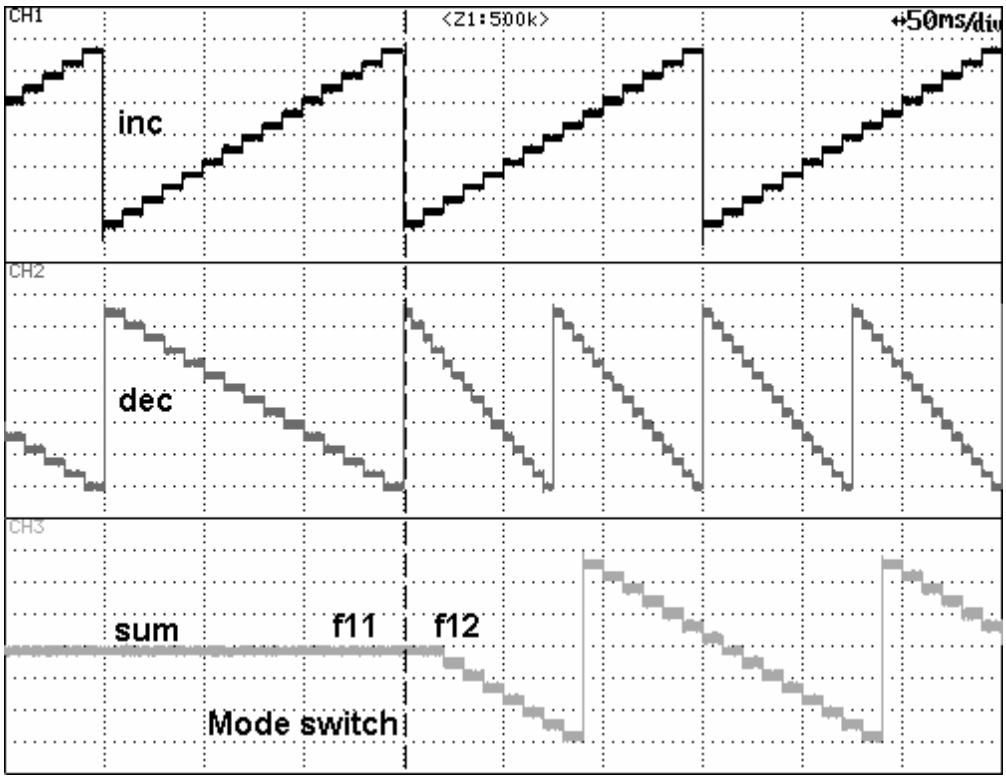


As a developer specifies only the timing behavior in TDL, the functionality of the tasks has to be implemented in another programming language. In this case study we assume that the functions invoked by the tasks and the drivers for reading sensors and updating actuators have been implemented in C as external functionality code. The TDL source code shown below indicates this according to the syntax of TDL by the keyword *uses*.

```
module M1 {

 public const
  c1 = 50; c2 = 200; refPeriod = 10ms;

 sensor
  int s uses getS;

 actuator
  int a1 := c1 uses setA1;
  int a2 := c2 uses setA2;

 public task inc {            // declaration of task inc
  output int o := c1;
  uses incImpl(o);   // inc. by step 10 (from 50 to 200)
 }

 public task dec {            // declaration of task dec
  output int o := c2;
  uses decImpl(o);    // dec. by step 10 (from 200 to 50)
 }

 // in the following we define the periodic activities of mode f11:
 //  (1) the invocations of the tasks inc and dec in this mode
 //  (2) the actuator updates
 //  (3) the frequency with which a mode switch is checked
 start mode f11 [period=refPeriod] {  // mode period is 10 ms
  task
    [freq=1] inc();  // LET of task inc is f11.period/inc.freq = 10ms
    [freq=1] dec(); // LET of task dec is f11.period/dec.freq = 10ms
  actuator
    [freq=1] a1 := inc.o;    // actuator a1 update every 10 ms
```

```
    [freq=1] a2 := dec.o;   // actuator a2 update every 10 ms
   mode
    [freq=1] if switch2f12(s) then f12;  //mode switch depends on s
  }

  mode f12 [period=refPeriod] {
   task
    [freq=1] inc();
    [freq=2] dec();    // LET of task dec is 10ms/2 = 5ms
   actuator
    [freq=1] a1 := inc.o;
    [freq=2] a2 := dec.o;   // a2 is updated every 5ms
   mode
    [freq=1] if switch2f11(s) then f11;  //mode switch depends on s
  }
}


module M2 {

  import M1;

  actuator
   int a := M1.c2 uses setA;

  public task sum {
   input int i1; int i2;
   output int o := M1.c2;
   uses sumImpl(i1, i2, o);  // o := i1 + i2;
  }

  start mode main [period=M1.refPeriod] {
   task
    [freq=1] sum(M1.inc.o, M1.dec.o);
   actuator
    [freq=1] a := sum.o;
  }
}
```

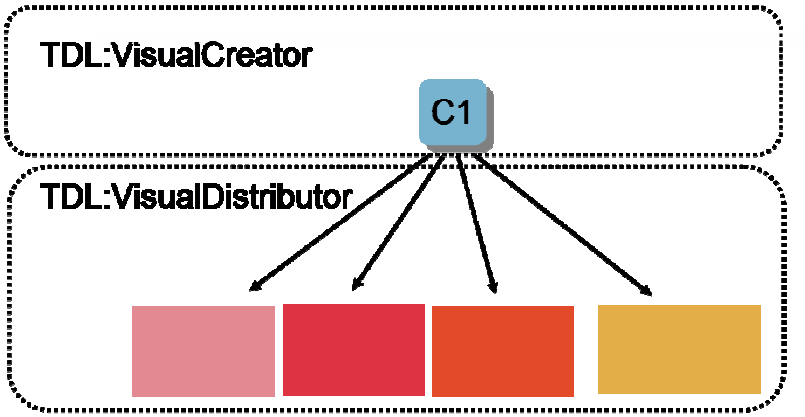
The following figure shows the outputs of module M1's inc and dec tasks and module M2's sum task. Module M1 is in mode f11 in the beginning (because the mode f11 is marked as start mode) so that the sum task is producing a constant output. After pushing the sensor button, a mode switch occurs and task sum produces the corresponding output pattern. The delay between the output of the sum task and the output of the inc and dec tasks is due to the LET semantics.



TDL modeling and platform deployment

This manual describes the platform-independent development of TDL components. Instead of the textual notation shown above, a visual/interactive tool can be used that we called the TDL:VisualCreator. We have seamlessly integrated the TDL:VisualCreator with Matlab/Simulink®. This offers the advantage that (1) the functionality of TDL components can be modeled with discrete blocks from the Simulink® library and (2) the TDL components can be simulated.

If you have your functionality implemented in C or other programming languages, the TDL:VisualCreator can also be used as stand-alone tool for visual/interactive TDL modeling, but without the simulation capability provided by Matlab/Simulink®. The next chapter guides you through the case study described above, with the TDL:VisualCreator in Matlab/Simulink®.



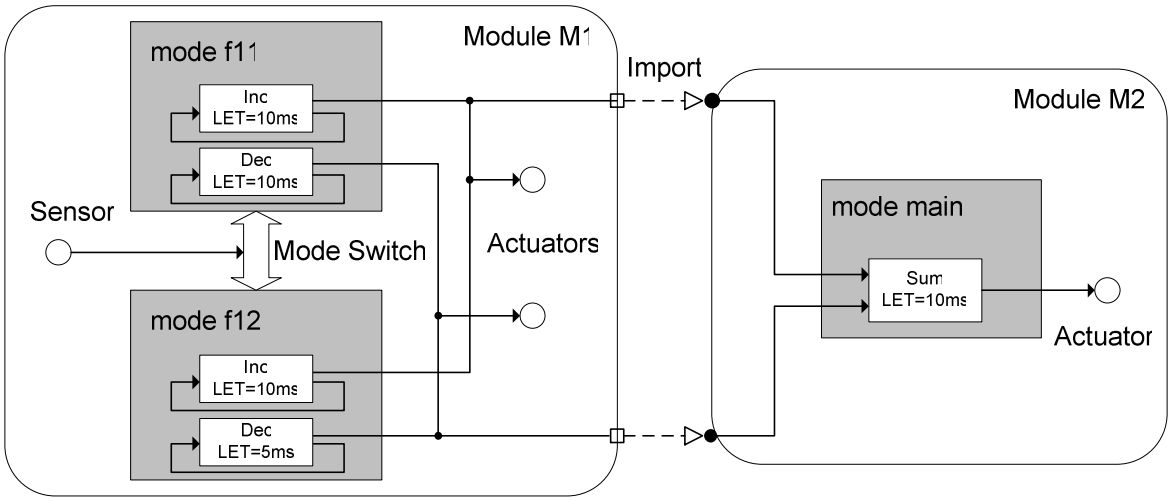
This manual does not cover the specification of the execution platform and the mapping of TDL components to such a platform, that is, the platform deployment. The socalled TDL:VisualDistributor tool offers both features, the platform modeling and platform deployment of TDL components.

For a description of the TDL:VisualDistributor tool we refer to its user manual which is not part of this documentation and not part of the TDL demo package.

Guided Tour of the TDL:VisualCreator tool

The guided tour illustrates the visual and interactive editing of two TDL components by means of the TDL:VisualCreator tool. We assume that you use the TDL:VisualCreator tool within the Matlab®/Simulink® environment.

As case study we stick to the two TDL sample modules introduced before. Remember the following figure which presents schematically the two modules with their modes, tasks, and ports.



Module *M1* has one sensor input, two tasks called *inc* and *dec*, and two actuators connected to the output ports of the tasks. The *inc* task increments its output value by 10, starting with the initial value 50 up to the upper limit 200. The *dec* task decrements its output value by 10, starting with the initial value 200 down to the lower limit 50. The sensor is only used for switching between the two modes of the module. In mode *f11* both tasks have the same LET, namely 10 ms. In mode *f12* the task *dec* has a LET of 5 ms—it produces the output values twice as fast as task *inc*.

Module M2 imports module M1 and thus has access to its public members, in particular to the output ports of M1's tasks *inc* and *dec*. Module M2's task *sum* simply adds the outputs of M1's *inc* and *dec* tasks. The LET of task *sum* is 10 ms.

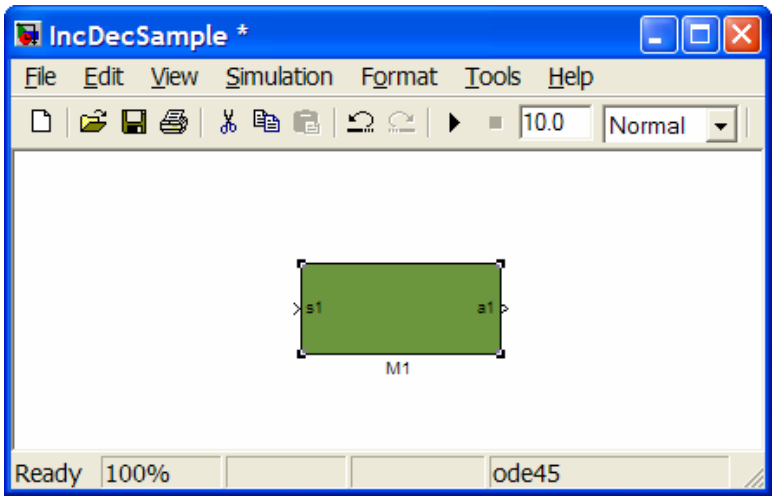
Define module *M1*

Module *M1* comprises many typical elements of a TDL module: the definition of constants, of a sensor and of two actuators as well as the definition of two tasks and two modes.

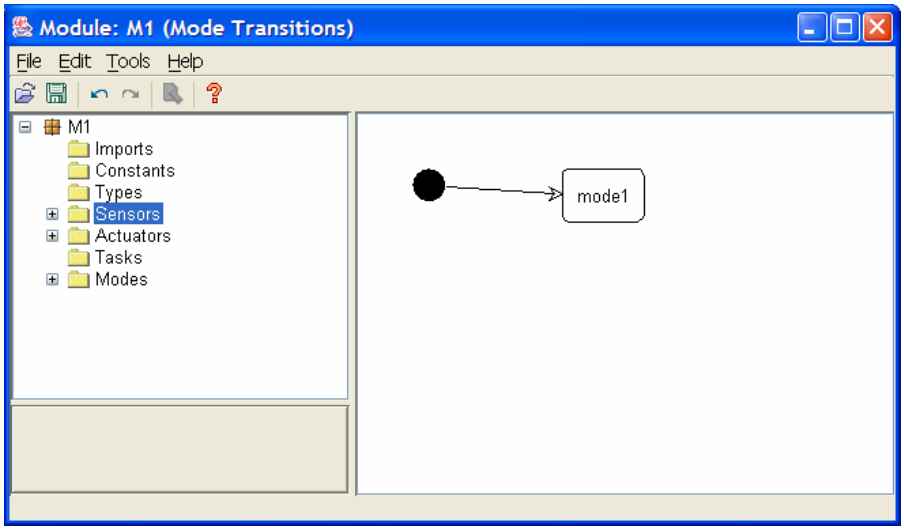
Start TDL:VisualCreator in Simulink®

First we start Matlab/Simulink® and open the library. (If the TDL Library with the green Module block does not appear as shown in the screen shot make sure that you set up Matlab/Simulink® properly. The yellow box is only listed if you also have licensed the TDL:VisualDistributor, which is not included in the demonstration version of the TDL:VisualCreator tool.)

_D2HLink_10162You drag the Module block from the *TDL Library* into a Simulink® editor (that is, a new model) and name the module *M1*. Save the Simulink® model as *IncDecSample*:



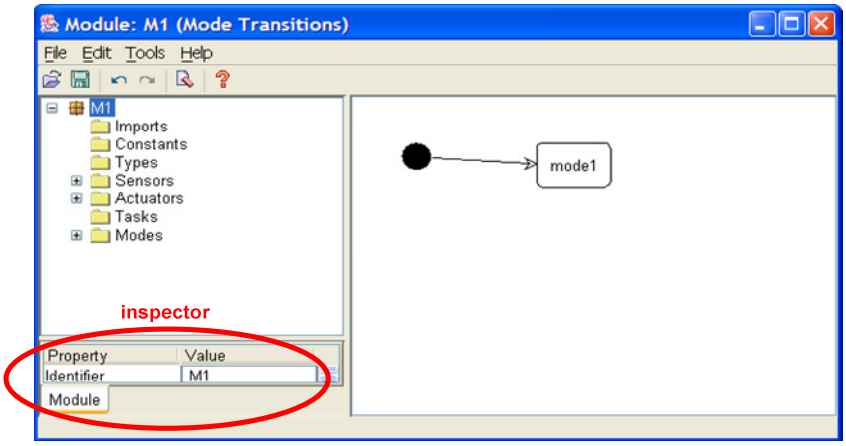
A double-click on the green box representing the TDL module M1 opens the TDL:VisualCreator tool:



The tree in the upper left window pane groups the module imports and TDL constructs of module M1 in folders. We call the tree pane just tree for short. The right-hand editor pane shows different editors depending on the editing state and your selection. The default editor is the Mode Transitions editor. A new module has as default mode one named *mode1* and that mode is marked as start mode by means of a bullet hole with an arrow pointing to the start mode. The empty window pane on the lower left side of the TDL:VisualCreator window is the property inspector view.

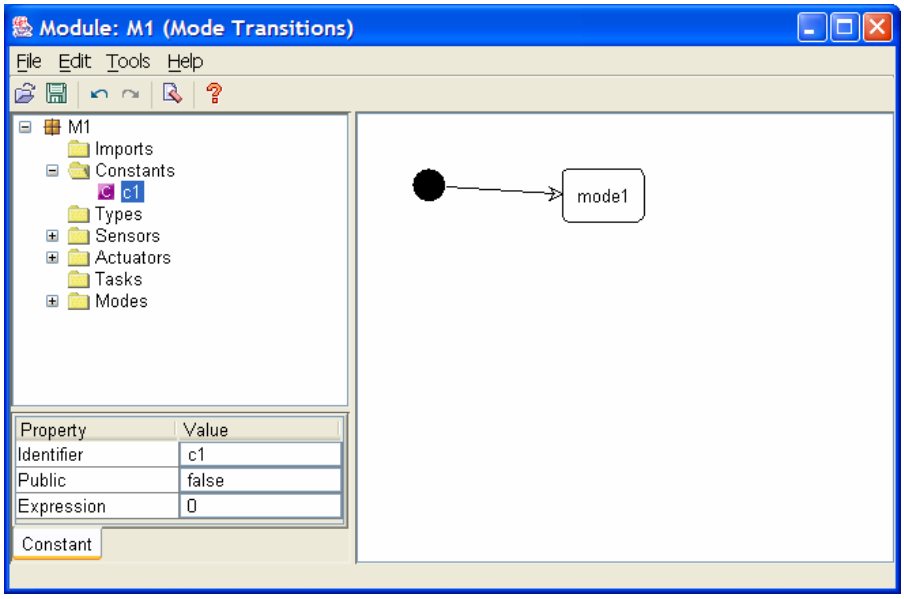
It shows the properties of items selected in the tree (see below). We call it inspector for short.

For example, select click on the module name M1. This displays its properties in the inspector in the window pane on the lower left side of the TDL:VisualDistributor tool. The only property of the module is the identifier property whose value corresponds to its name:

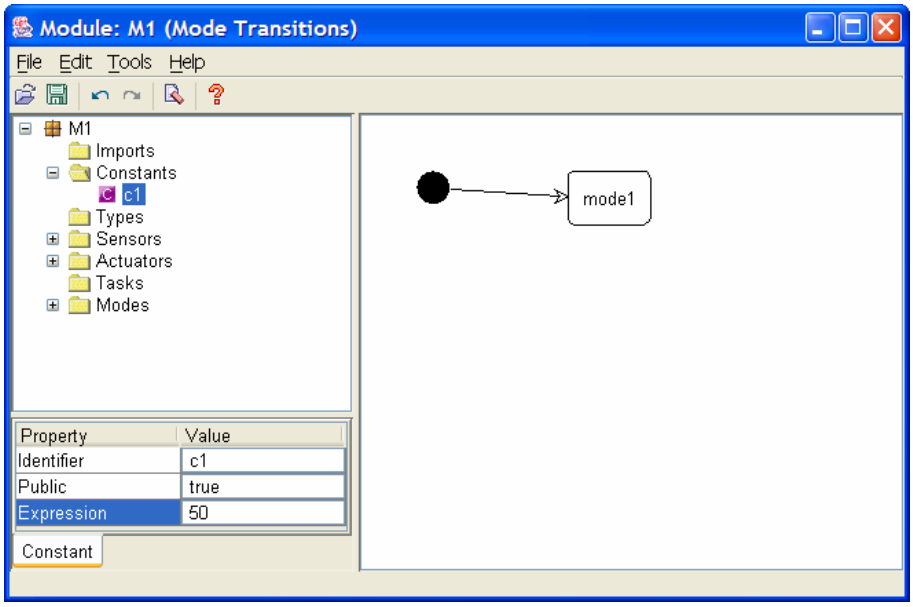


Add constants *c1* and *c2*

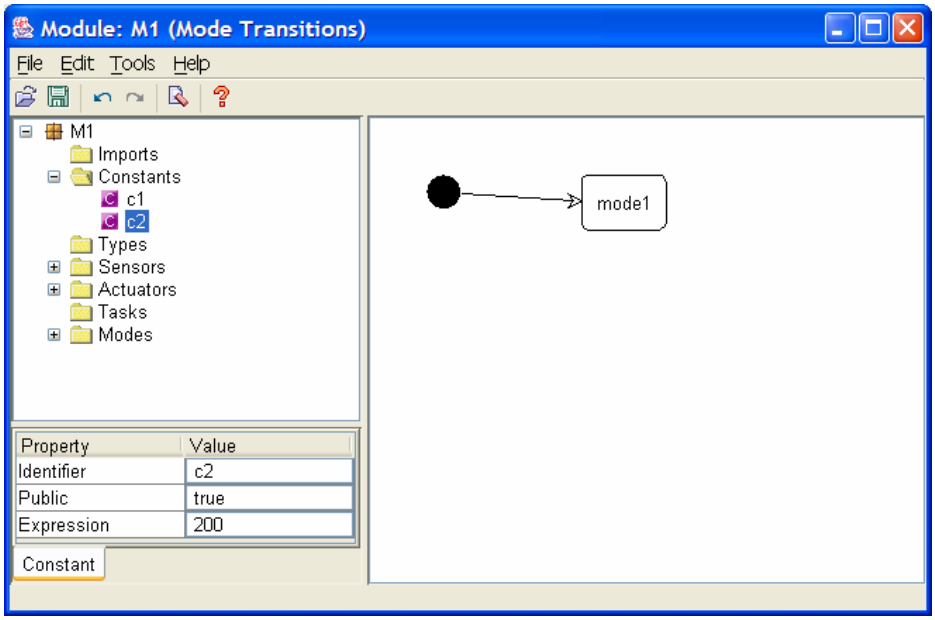
To add the constant *c1* choose the File>New>Constant menu item (that is, the menu item Constant in the New submenu of the File menu) or right-click on the *Constants* folder in the tree and select New Constant from the pop-up menu. We stick to the suggested default name *c1*. Click on the constant *c1*. As we have seen above with the module M1, selecting an item in the tree displays its properties in the inspector, that is, in the window pane below the tree:



In the inspector mark the access as public (that is, set the value of the Public property to true), and set the Expression to 50:

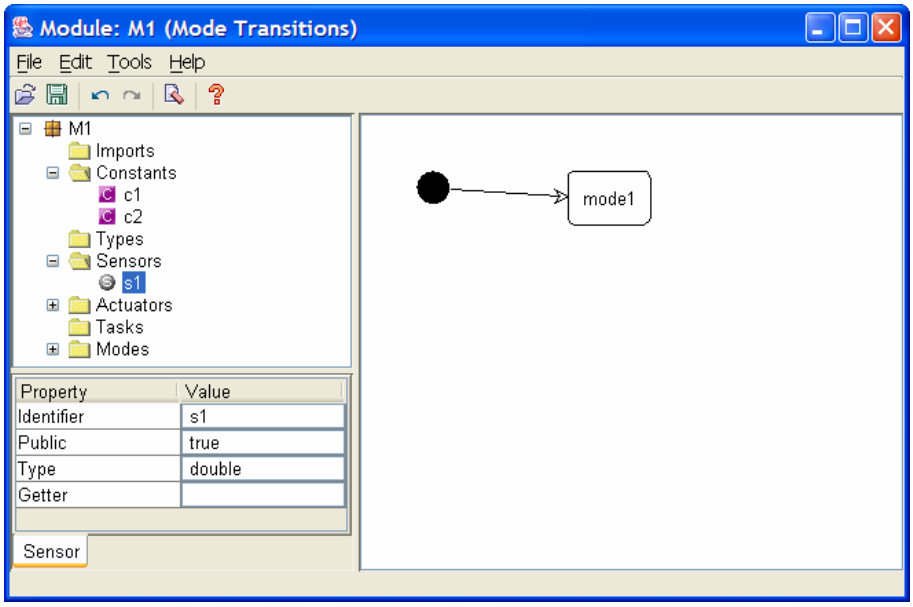


To add the constant *c2* choose again the File>New>Constant menu item or right-click on the *Constants* folder in the tree. We stick to the suggested default name *c2*. Mark the access property of constant *c2* as public (that is, set the value of the Public property to true), and set the Expression to 200:

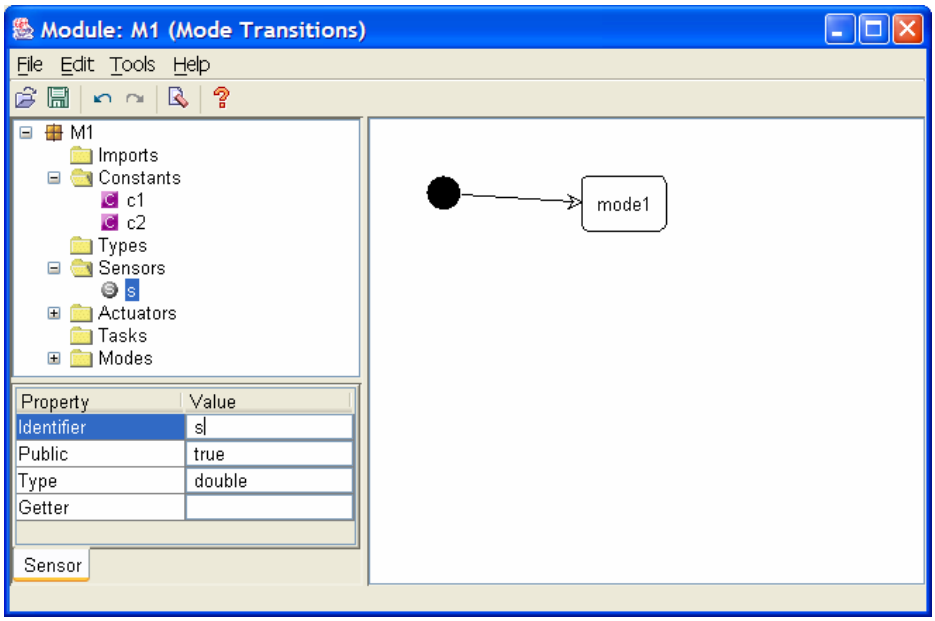


Edit sensor *s*

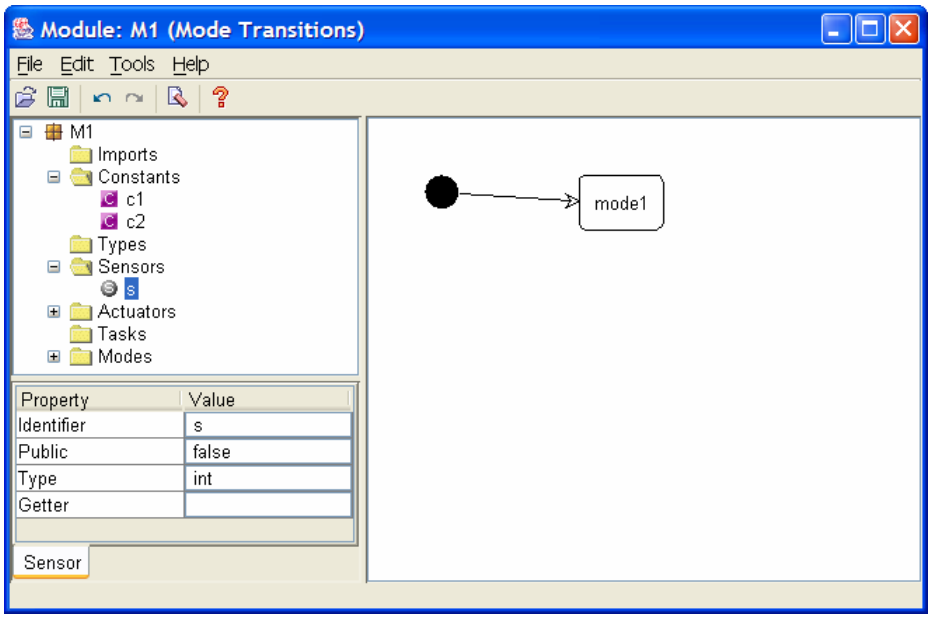
Each new module has already one sensor. In order to rename this sensor open the *Sensors* folder in the tree (click on the + symbol left to the *Sensors* folder in the tree) and click on the sensor labeled s1:



Either choose the Edit>Rename menu item (that is, the menu item Rename in the Edit menu) or right-click on the listed *s1* and choose Rename from the pop-up menu. Rename the sensor from *s1* to *s*. You can also click twice (no double-click speed!) on the *s1* item to edit the name directly in the tree. Another option is to edit the Identifier property in the inspector.

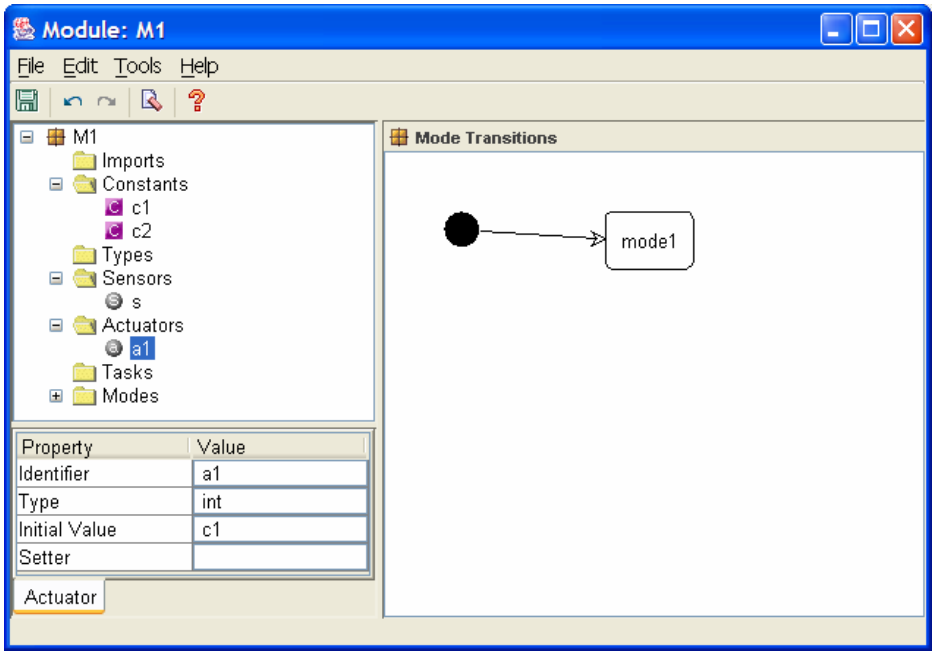


In the inspector change the sensor type to integer (int) and make it private (set the Public property to false):

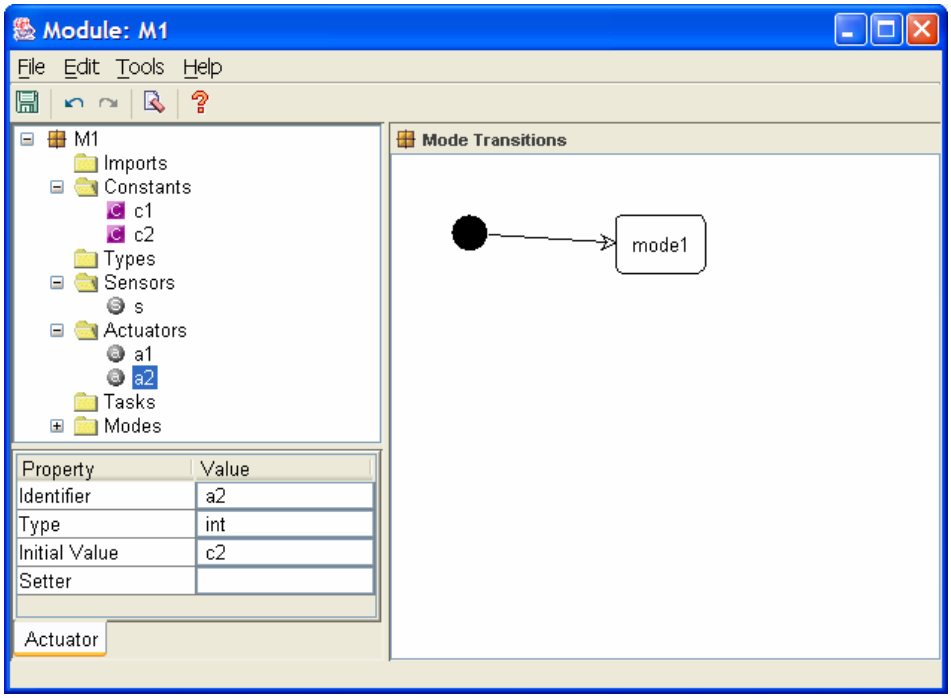


Edit actuators *a1* and *a2*

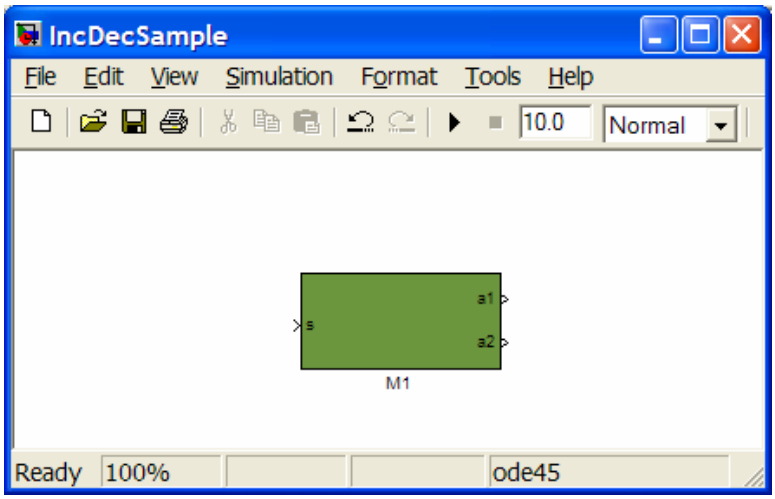
Each new module has already one actuator. We stick to the suggested default name *a1*. In the inspector change the Type property to int, and set the Initial Value to c1 by selecting that constant from the pop-up menu.



In an analogous way edit actuator *a2*. Choose the File>New>Actuator menu item or right-click on the *Actuators* folder in the tree. We stick to the suggested default name *a2*. Change the Type property to int, and set the Initial Value to *c2* by selecting that constant from the pop-up menu.

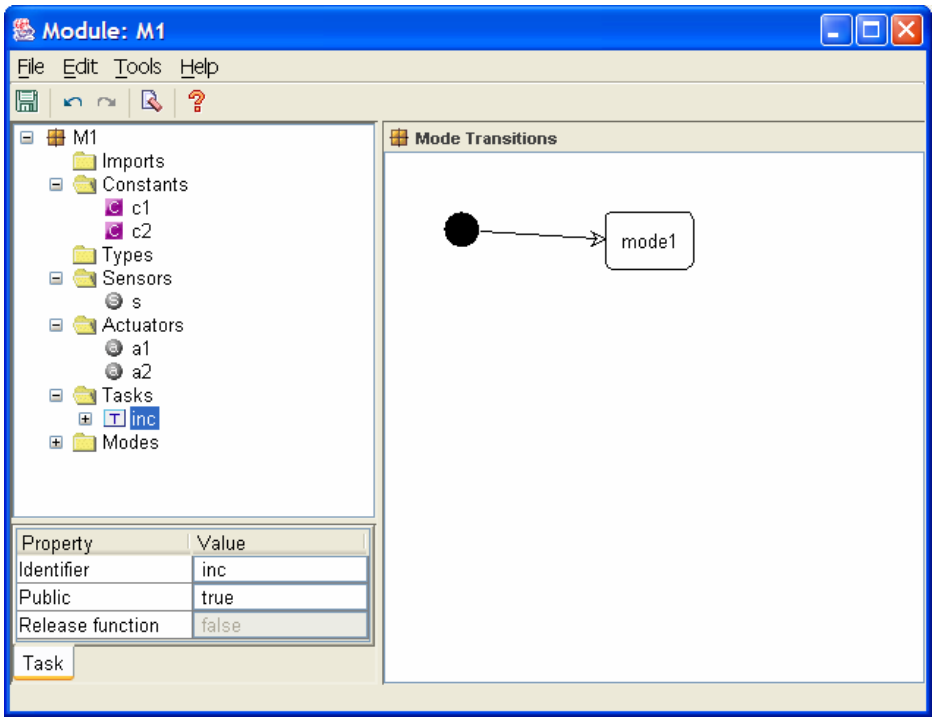


The editor window which contains module *M1* also shows that *M1* has one inport *s* and two outports *a1* and *a2*:

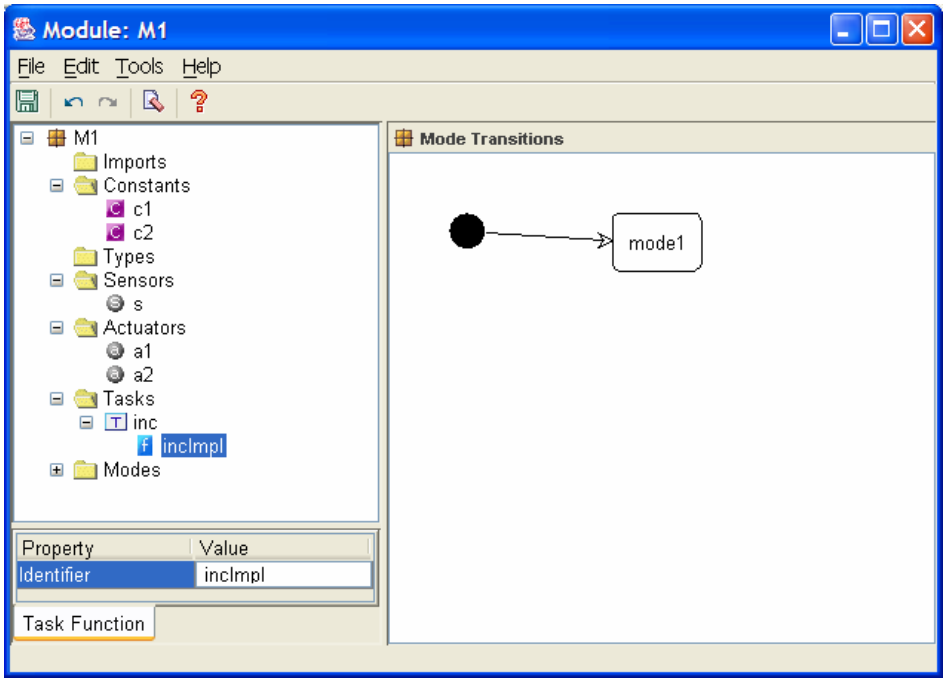


Edit tasks *inc* and *dec*

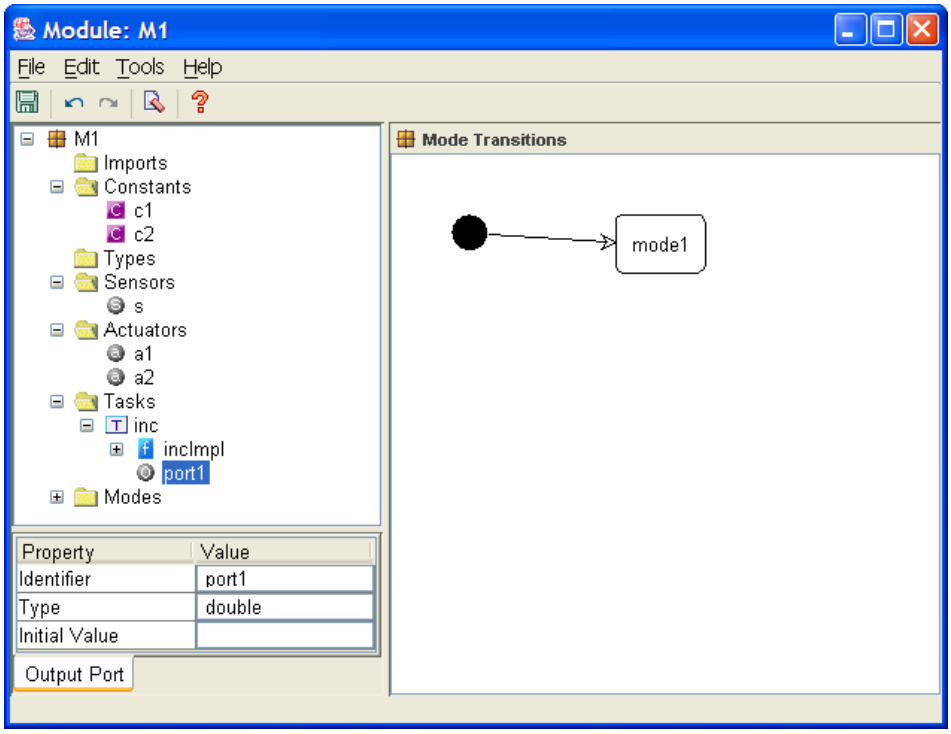
From the File>New submenu choose Task or right-click on the *Tasks* folder in the tree and choose the New Task item from the pop-up menu. Analogous to editing constant, sensor and actuator names, change the name to *inc*, and make the access public (set the Public property to true). Ignore for now the 'Release function' property, which indicates whether the task is part of a task sequence or not. The concept of task sequences that allow the updating of actuators in logical zero time is explained in the section about synchronous actuator updates.



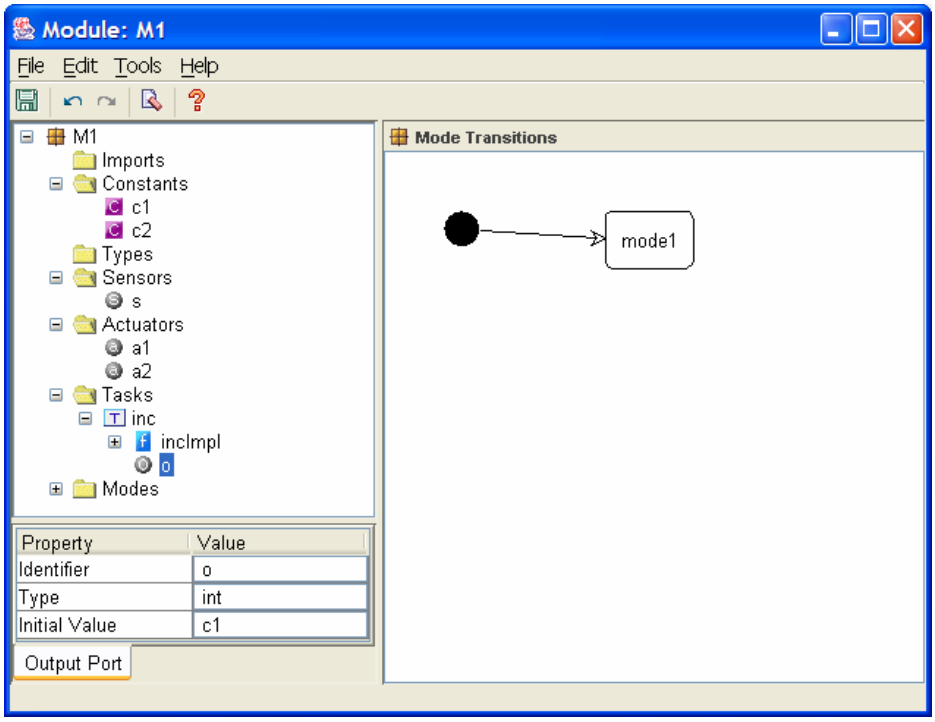
Open the *inc* folder in the tree (click on the + symbol left to the *inc* folder in the tree; the *inc* folder is a subfolder of *Tasks*) and rename the item labeled *t1Impl* to *incImpl*. The *incImpl* item refers to the task function implementation.



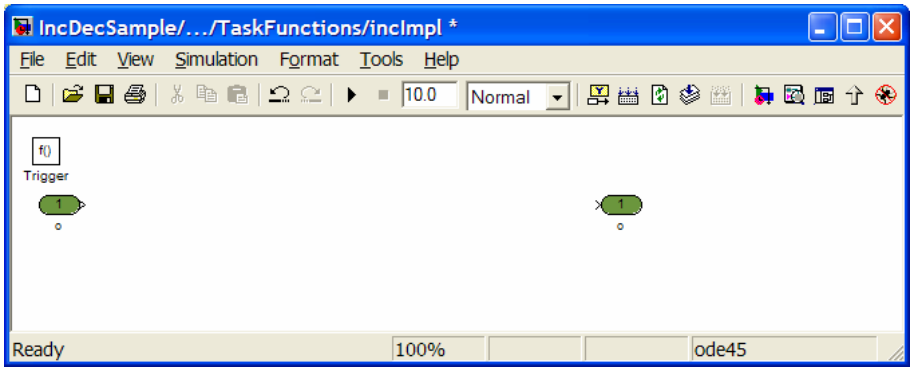
Right-click on the *inc* task (not the *incImpl* function) in the Tasks folder in the tree and choose the New ouput port menu from the pop-up menu. This adds an output port labeled *port1* to the task declaration of task *inc*:



Rename the output port to *o*, and change its type to integer (int) with the Initial Value of *c1* (select the constant from the pop-up menu):

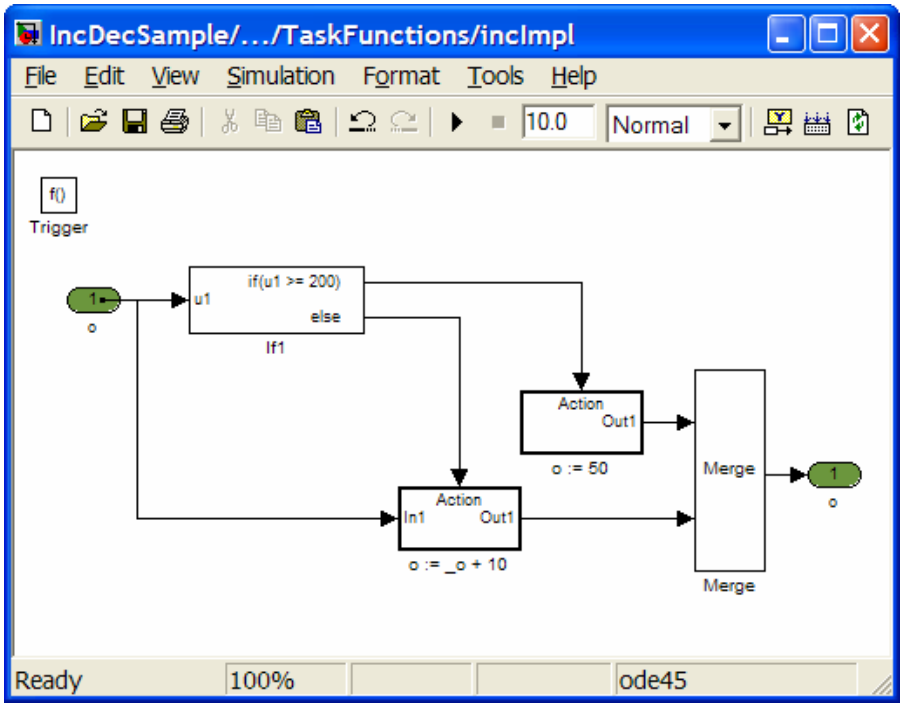


The task function can now be modeled in Simulink® by simply double-clicking on the *incImpl* item in the tree. This opens a Simulink® editor window:

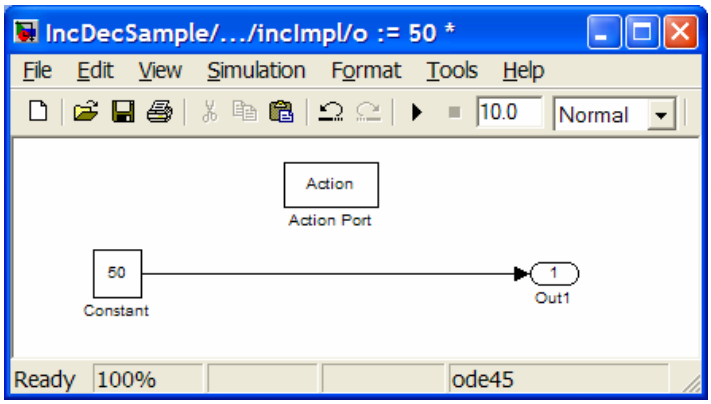


The outport *o* on the right side in the window corresponds to the output port of the task. The inport *o* provides the value of *o* as calculated in the previous task invocation.

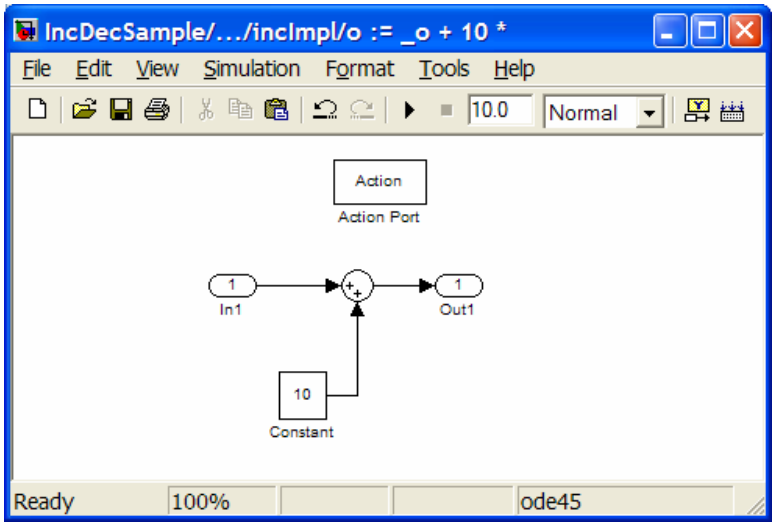
We model the straight-forward functionality of task *inc* by means of standard Simulink® blocks: The *incImpl* increments its output value by 10, starting with the initial value of *c1* (50) up to the upper limit *c2* (200):



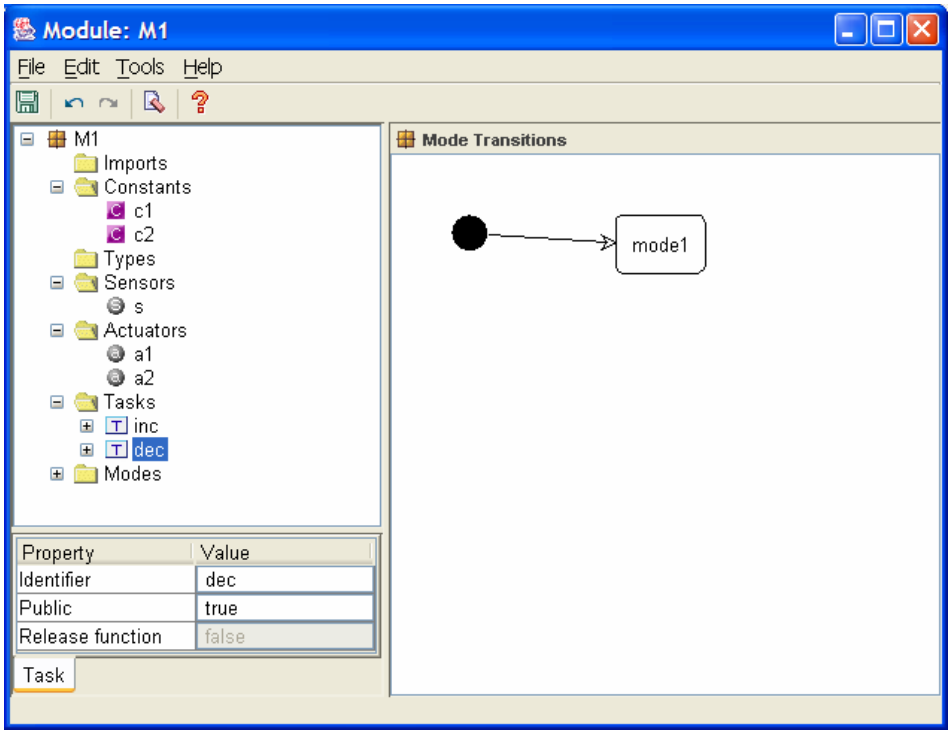
The If-Action-block labeled o := 50 contains the following blocks:



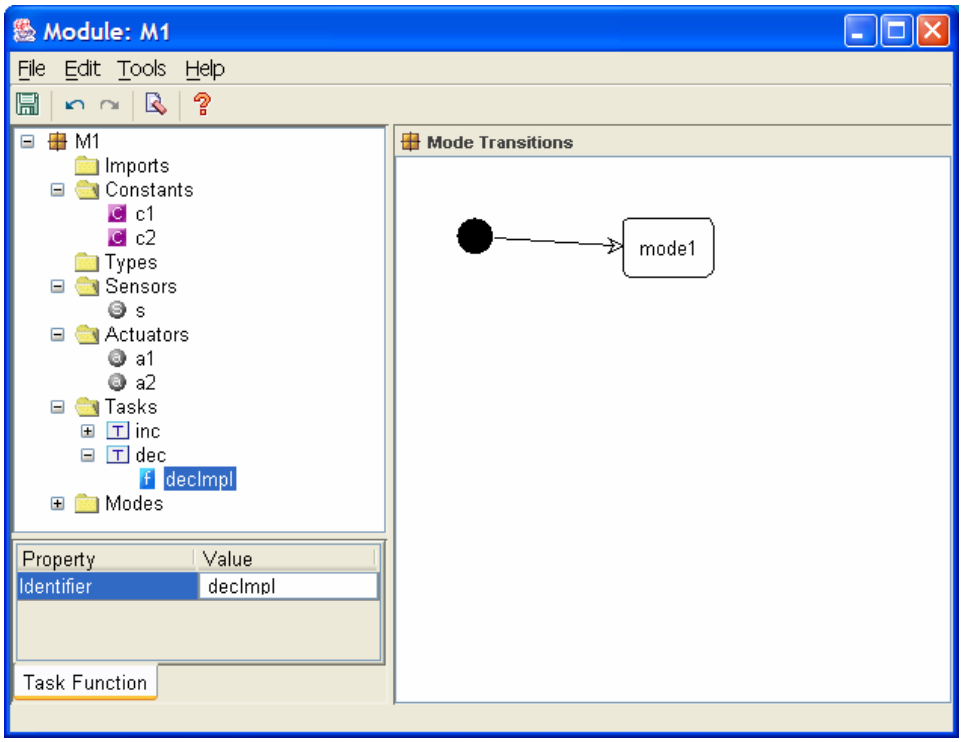
The If-Action-block labeled o := _o + 10 contains the following blocks:



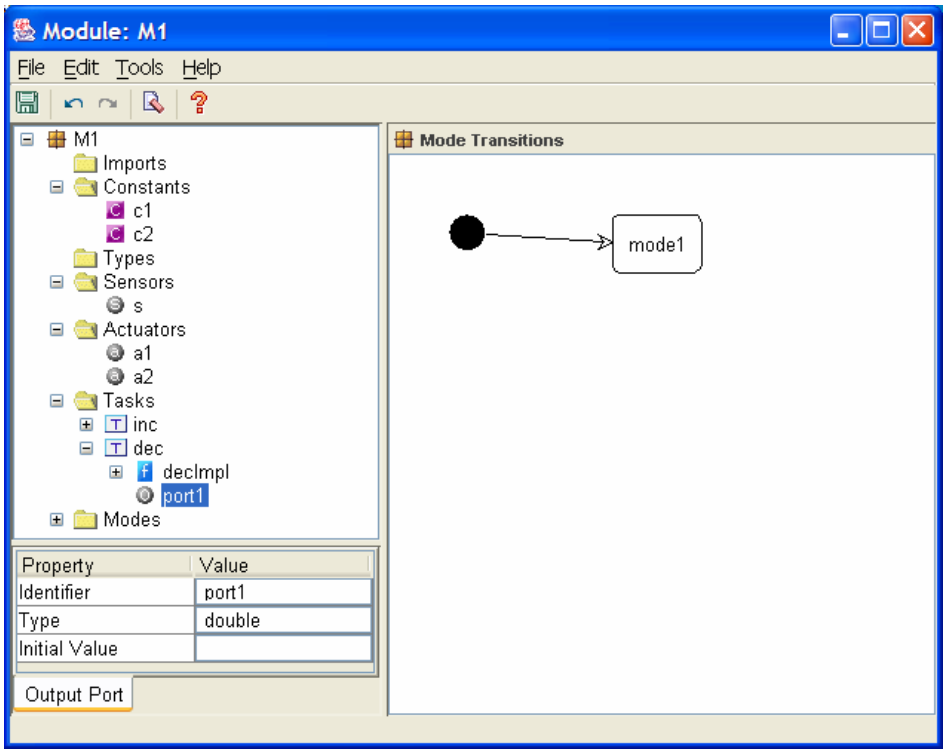
Task *dec*. We define task *dec* in an analogous way as task *inc*. From the File>New submenu choose Task or right-click on the *Tasks* folder in the tree pane and choose the New Task item from the pop-up menu. Change the name to *dec*, and make the access public (set the Public property to true).



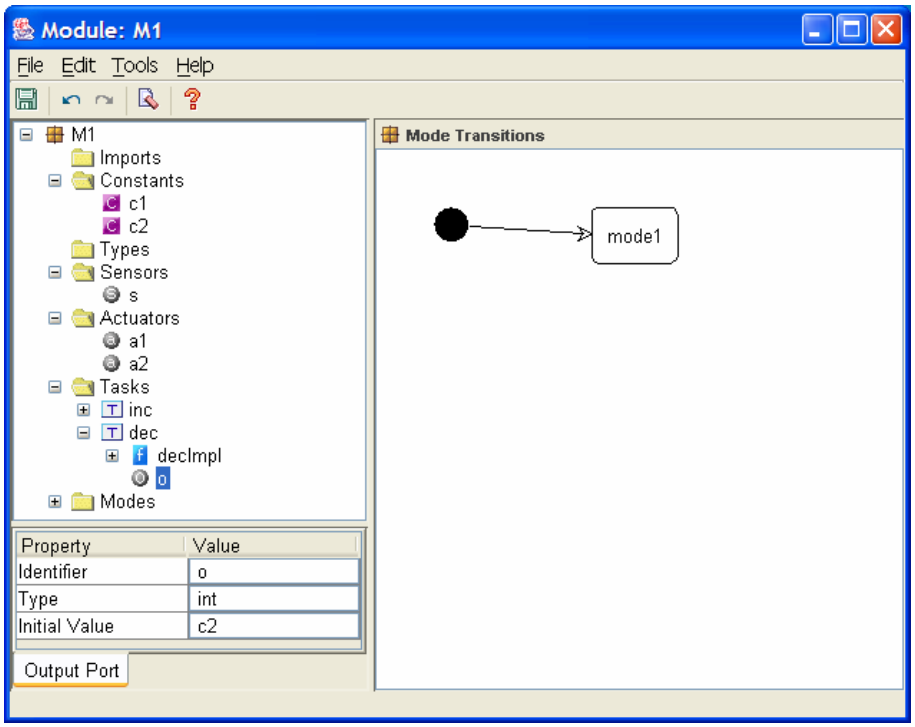
Open the *dec* folder in the tree (click on the + symbol left to the *dec* folder in the tree; the *dec* folder is a subfolder of *Tasks*) and rename the item labeled *t1Impl* to *decImpl*. The *decImpl* item refers to the task function implementation.



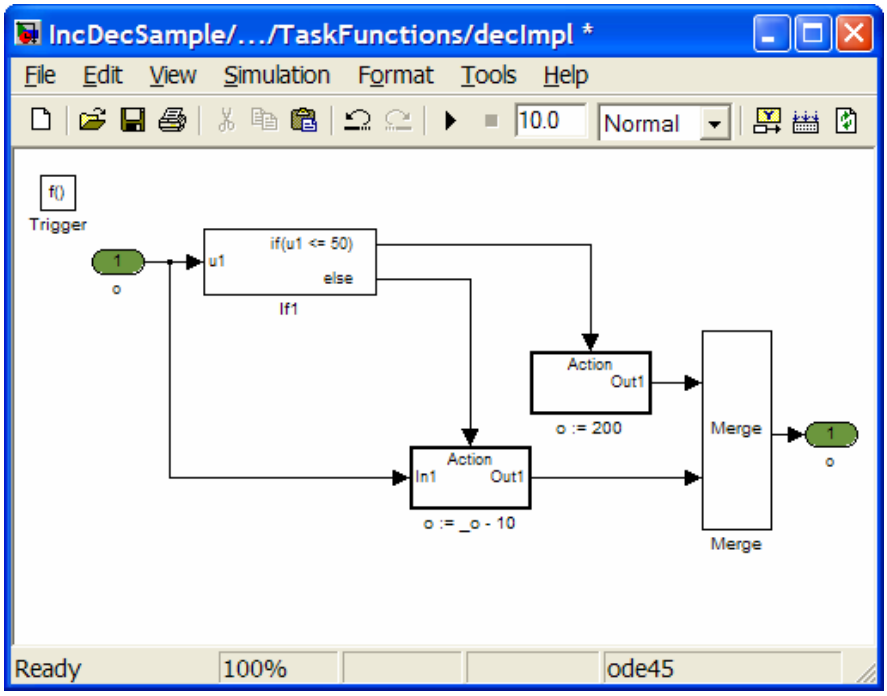
Right-click on the *dec* task (not the *decImpl* function) in the Tasks folder and choose the New ouput port menu from the pop-up menu. This adds an output port labeled *port1* to the task declaration of task *dec*:



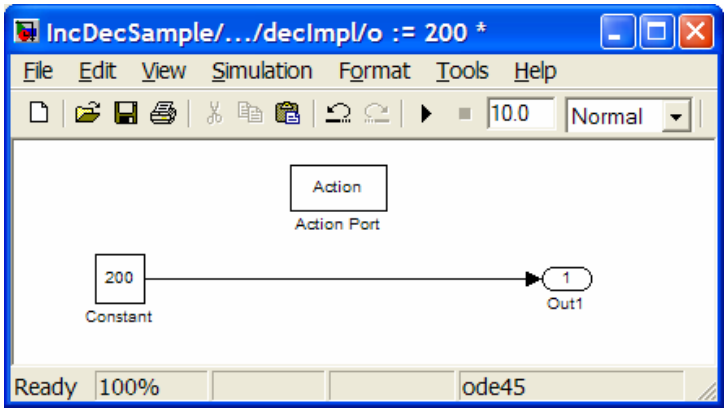
Rename the output port to *o*, and change its type to integer (int) with the Initial Value of *c2* (select the constant from the pop-up menu):



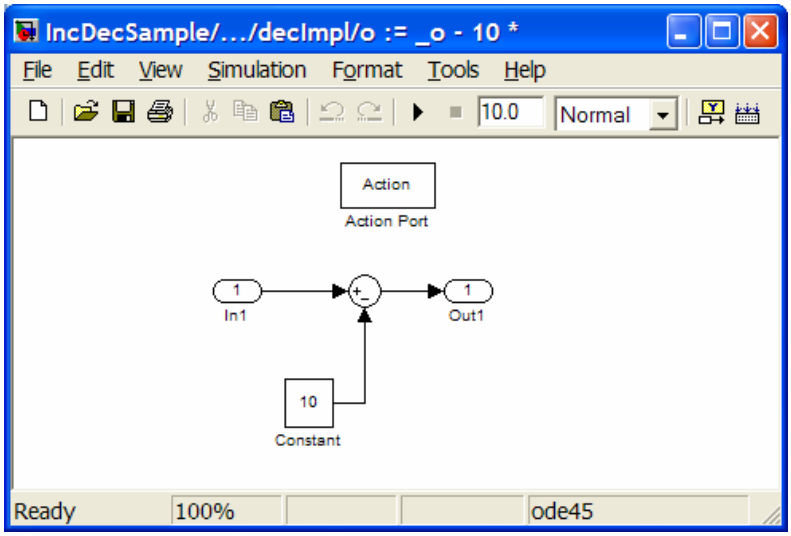
We model the task function again in Simulink® by simply double-clicking on the *decImpl* item. The *decImpl* decrements its output value by 10, starting with the initial value 200 down to the lower limit 50:



The If-Action-block labeled o := 200 contains the following blocks:

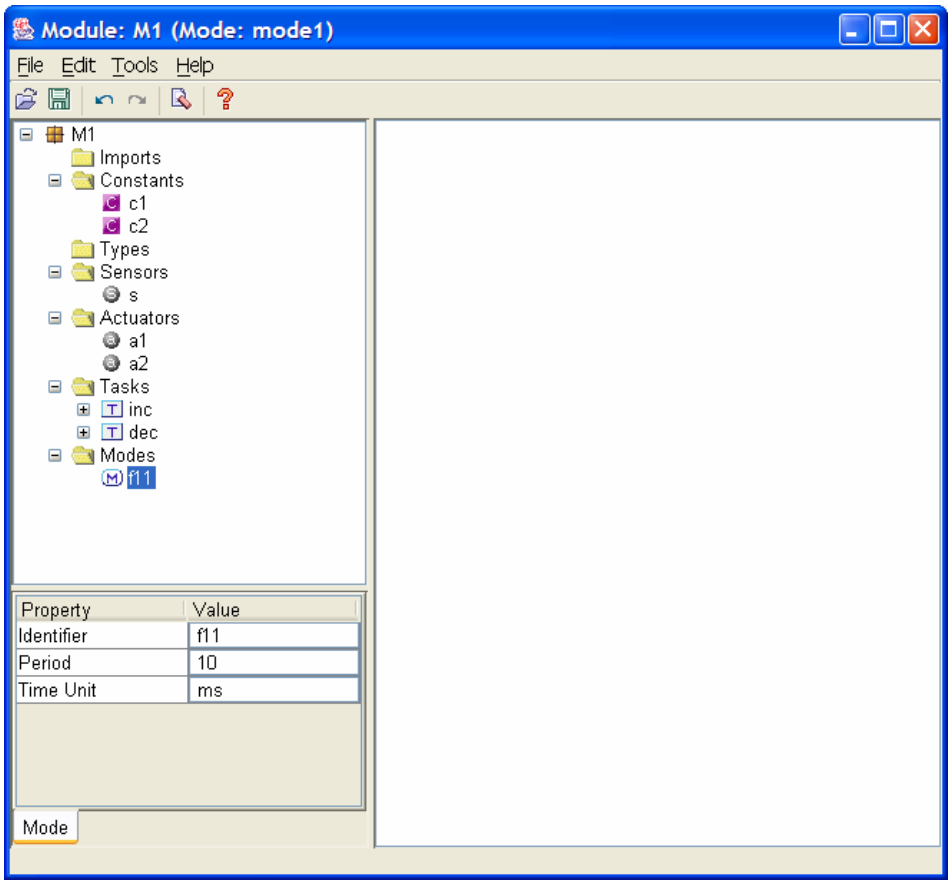


The If-Action-block labeled o := _o - 10 contains the following blocks:

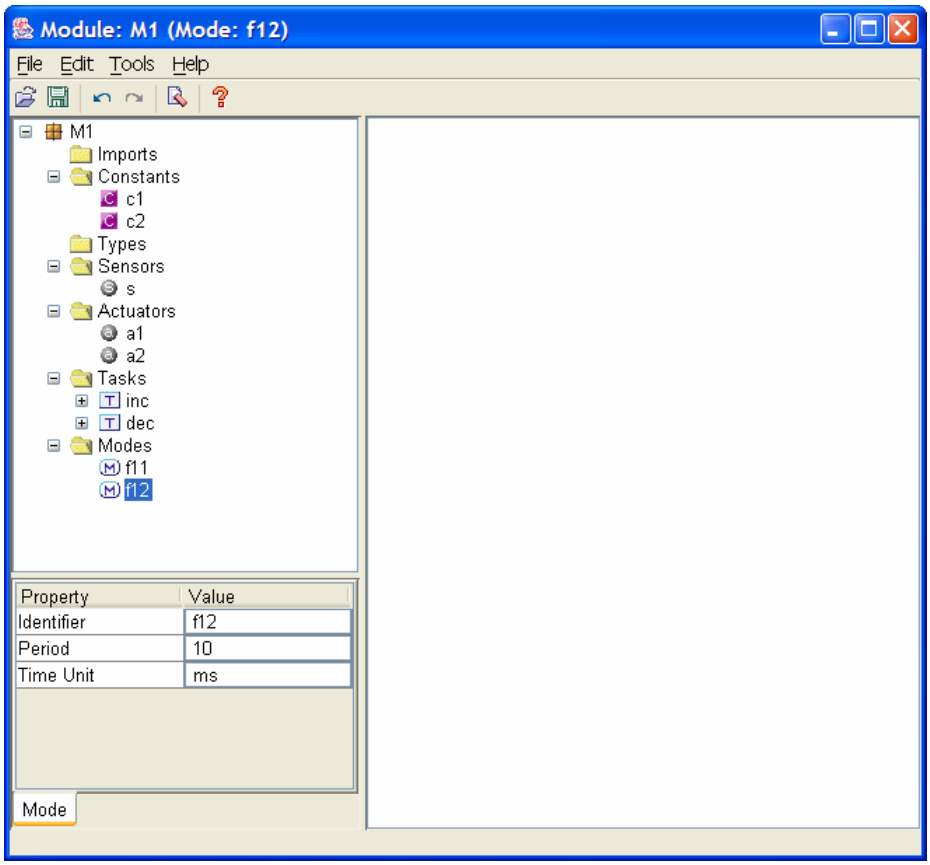


Edit modes *f11* and *f12*

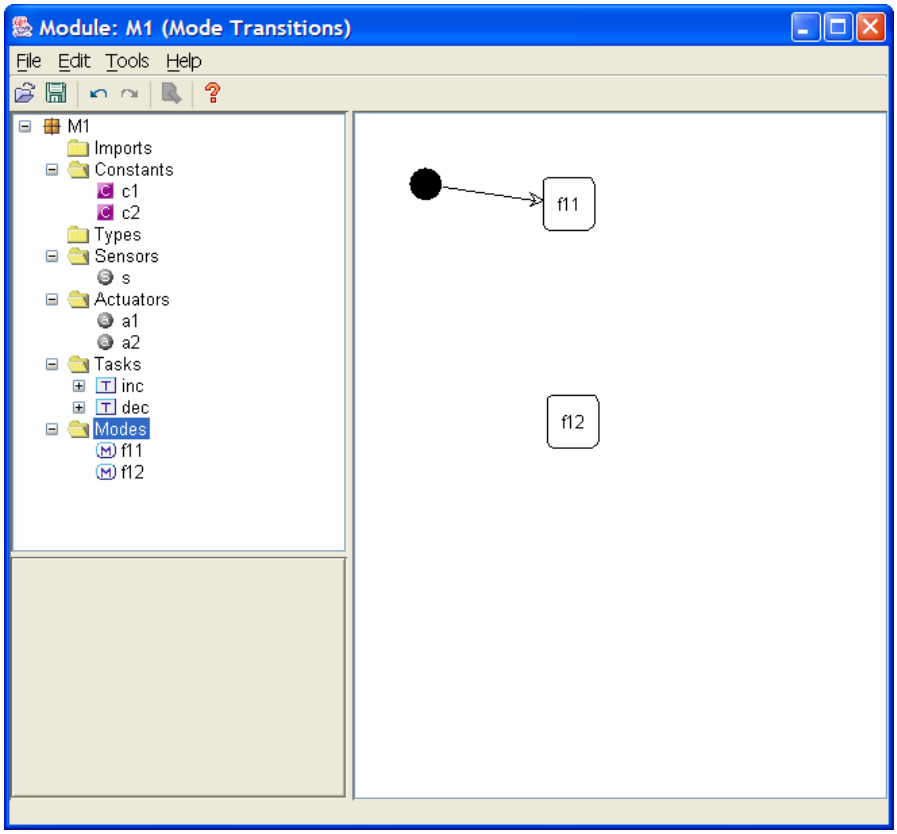
Each new module has already one mode called *mode1*. In order to rename this mode open the *Modes* folder (click on the + symbol left to the *Modes* folder in the tree). Click twice (no double-click) on the *mode1* item and change the name to *f11*. Alternatively, you can right-click on the *mode1* text in the tree and choose the Rename item from the pop-up menu. You can also change the name by selecting the *mode1* text in the tree. Then you edit the name in the Identifier property of the inspector. Set the Period property to 10 and leave the Time Unit property as milliseconds (ms):



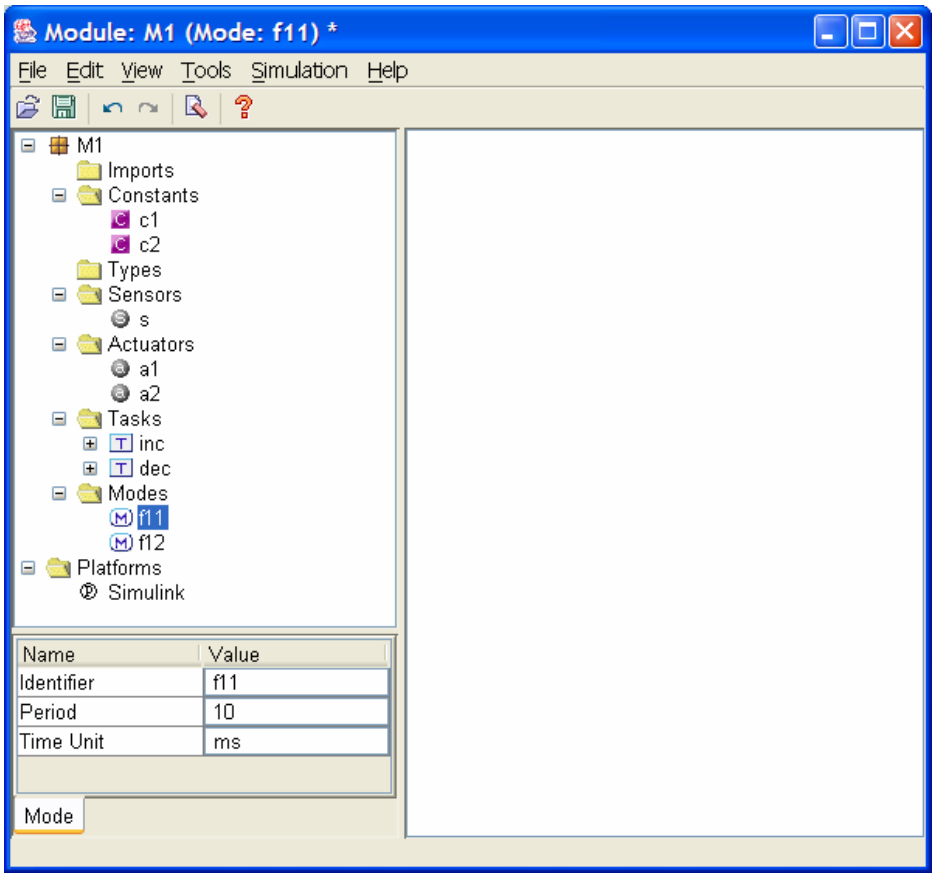
Next we add mode *f12* by choosing the File>New>Mode menu item or by right-clicking on the *Modes* folder in the tree pane and choosing the New Mode item from the pop-up menu. Rename the new mode to *f12* as described above. Set the Period property to 10 and leave the Time Unit property as milliseconds (ms):



Select the Modes folder in the tree so that the Mode Transitions editor is activated in the right window pane. Note that the arrow from the black bullet hole to the box representing mode *f11* marks mode *f11* as start mode:

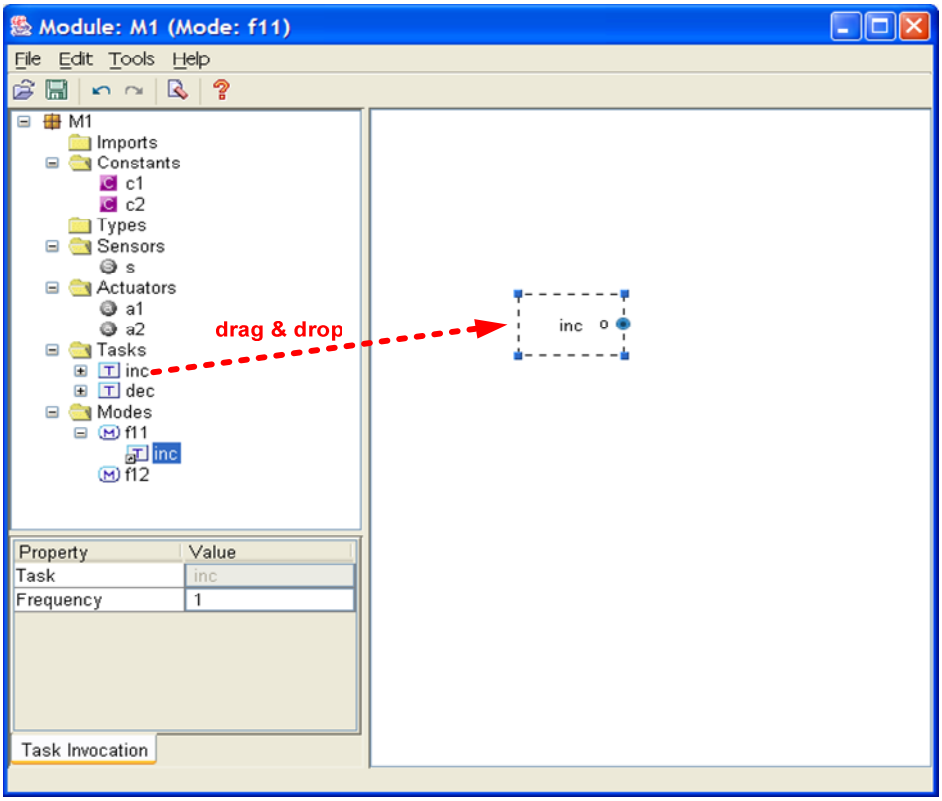


Task invocations and actuator updates. To assign tasks to modes we activate the Mode editor by selecting the particular mode. Let us start with *f11*:

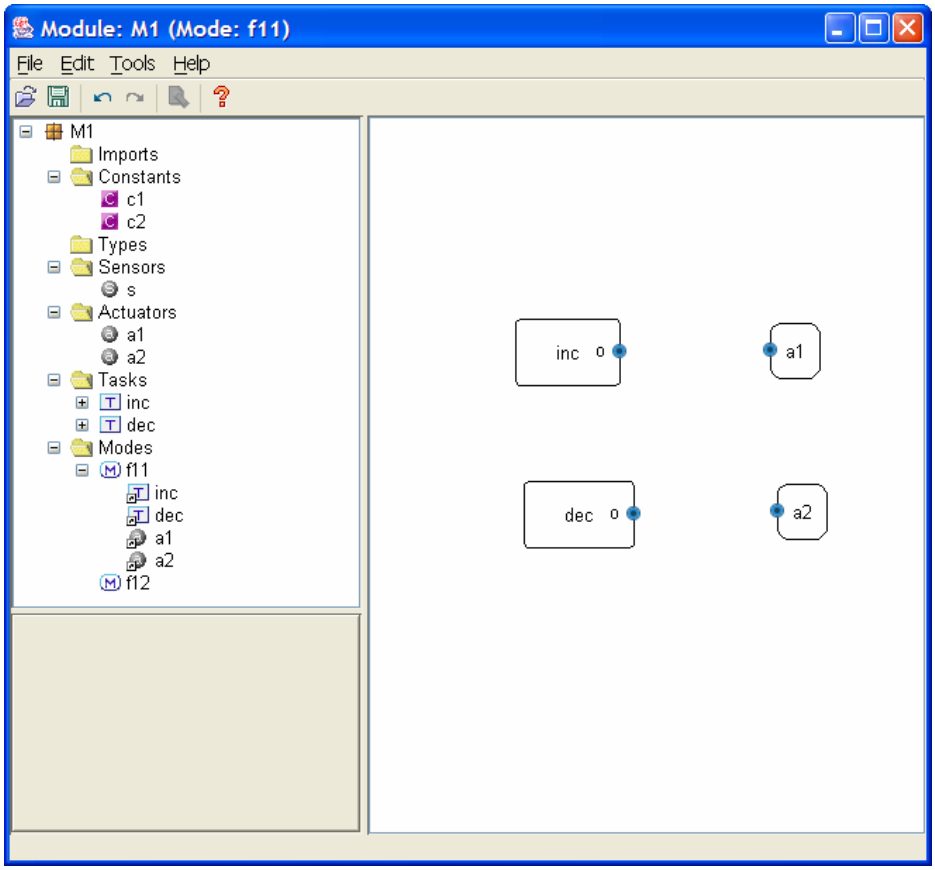


Note that you can switch back to the Mode Transition editor by either selecting the *Modes* folder or the module name (*M1* in our case study) in the tree. For now we keep the Mode editor active to edit mode *f11*. We drag the task *inc* in the Tasks folder and drop it on the Mode editor (right window pane). The operation drag means that you select the inc item in the tree with a left-click (left mouse button), but you do not release the mouse button. Then you move the item while keeping the left mouse button pressed. The drop operation means that you release the left mouse button at the point where you want to drop the dragged item. In this case this is somewhere in the Mode editor.

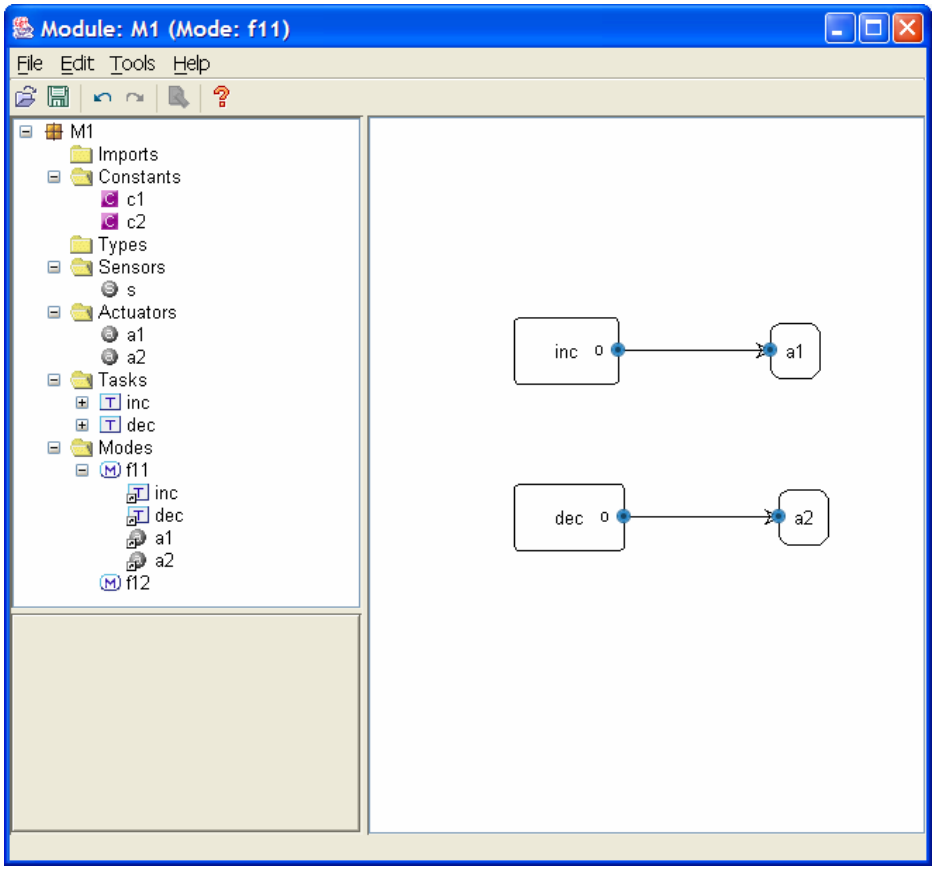
The invocation frequency of a task relative to the mode period is specified as the Frequency property. As the mode period of *f11* is 10 ms this means that *inc* has a Logical Execution Time (LET) of 10/1 = 10 ms.



Next we drag and drop task *dec* as well as the actuators *a1* and *a2* from the tree into the Mode editor:

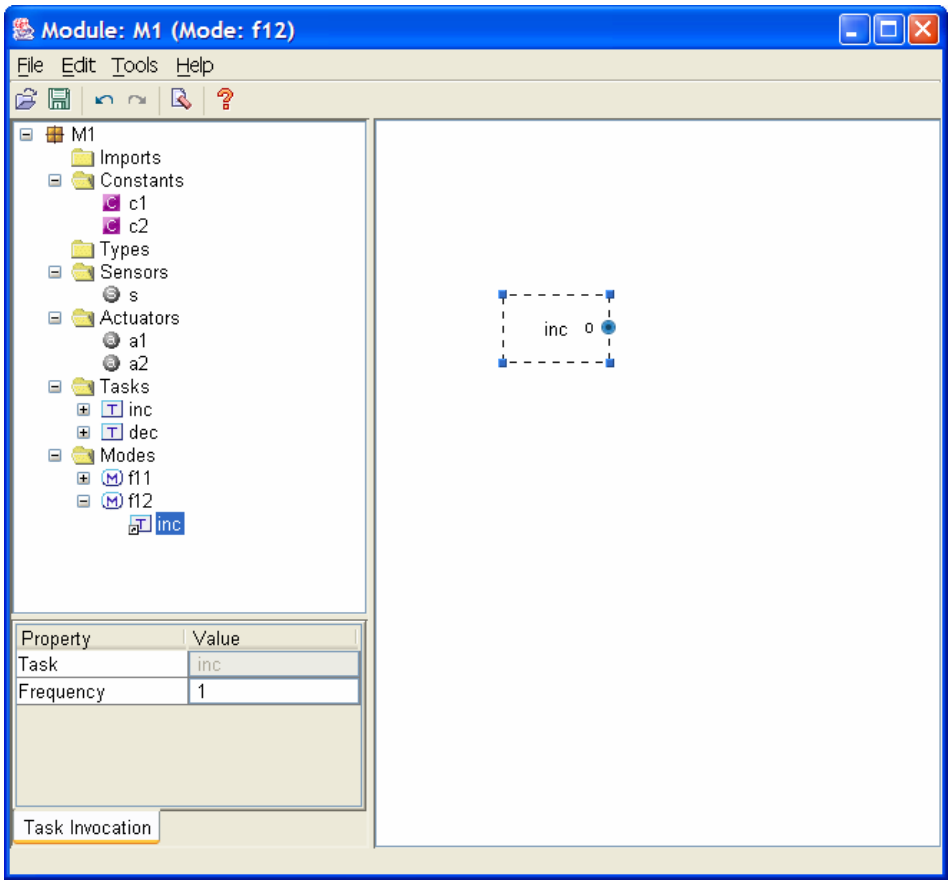


Then we connect the output port of task *inc* with actuator *a1*. This corresponds to the assignment a1 := inc.o in the textual version of a TDL program. In an analogous way we connect the output port of task *dec* with actuator *a2*:

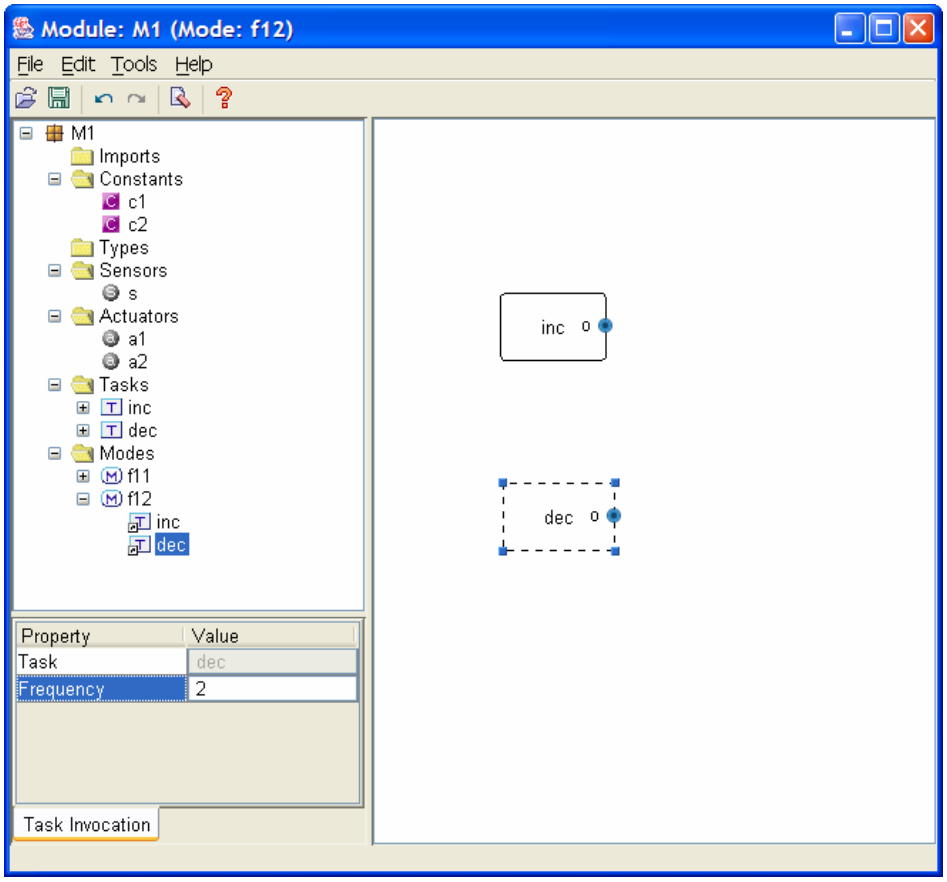


Mode *f12* is defined in an analogous way as mode *f11* with the only differences that the invocation frequency of task *dec* is 2 and that the actuator update frequency of *a2* is also 2. Let us define *f12* step by step:

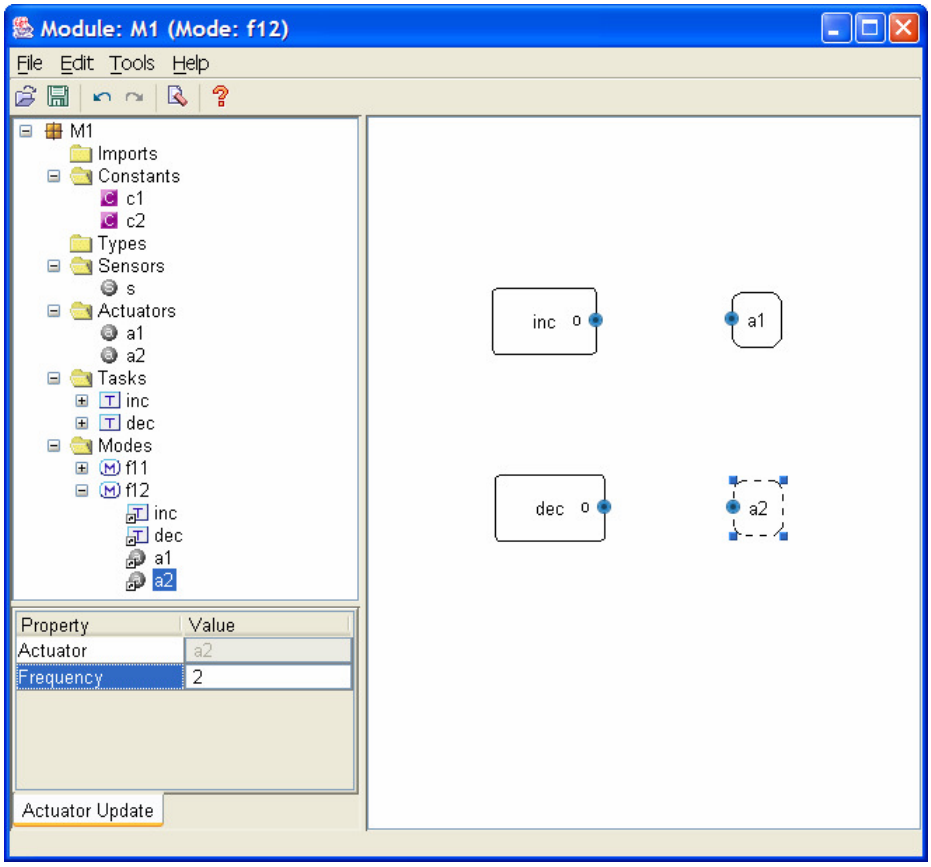
We select mode *f12* in the tree to activate its Mode editor. Then drag the task *inc* from the Tasks folder in the tree and drop it on the Mode editor (right window pane). We leave the Frequency property of task *inc* set to 1, which means that its LET in mode *f12* is also 10 ms as in the other mode *f11*.



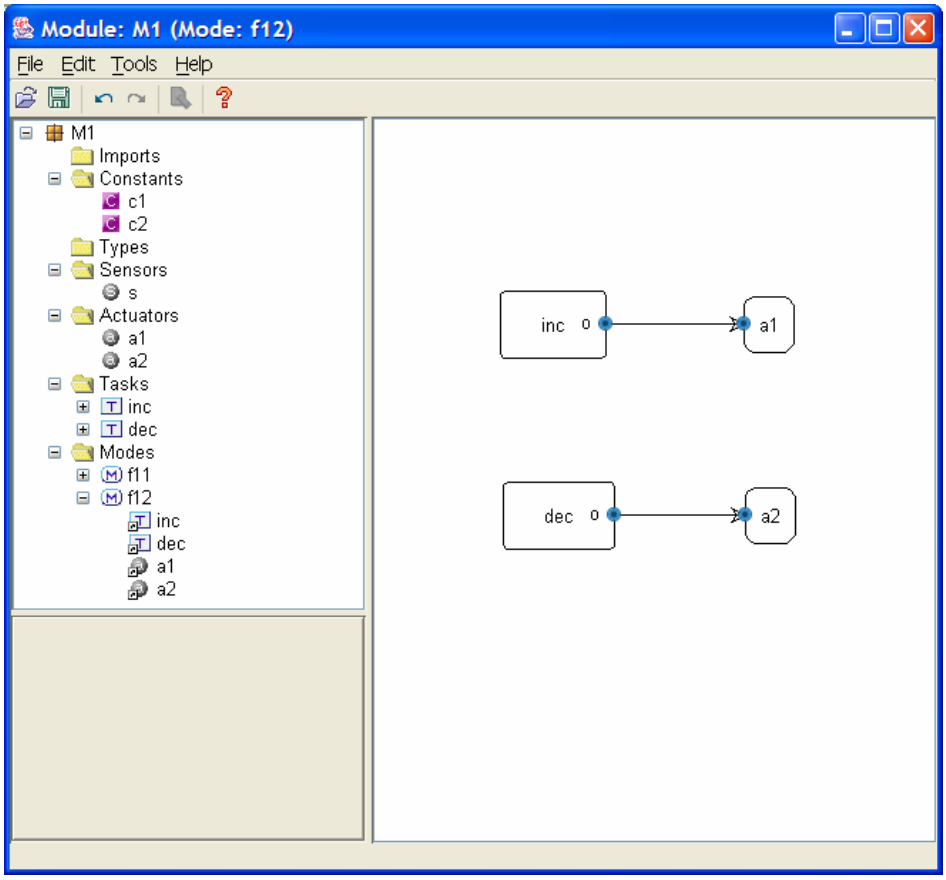
Next drag the task *dec* from the Tasks folder in the tree and drop it on the Mode editor. Set the Frequency property of task *dec* to 2, which means that its LET is 10 ms / 2 = 5 ms.



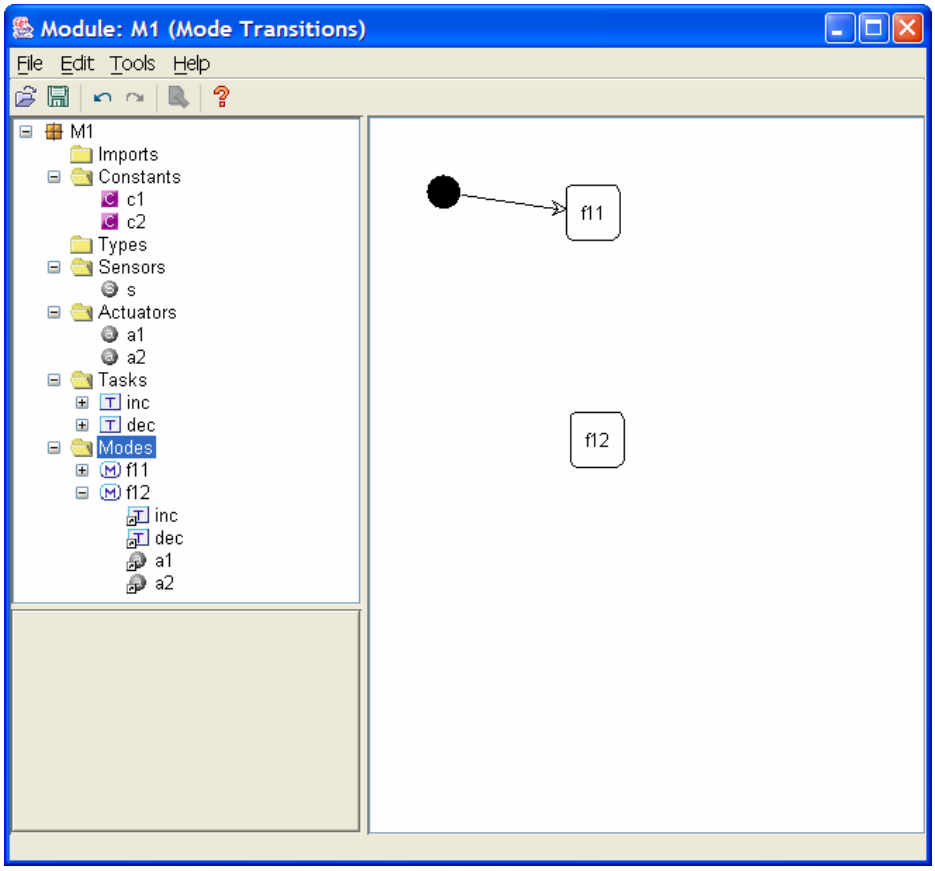
Next we drag and drop the actuators *a1* and *a2* from the tree into the Mode editor. We set the Frequency property of *a2* to 2 and leave the Frequency property of *a1* set to 1.



Then we connect the output port of task *inc* with actuator *a1*. This corresponds to the assignment a1 := inc.o in the textual version of a TDL program. In an analogous way we connect the output port of task *dec* with actuator *a2*:



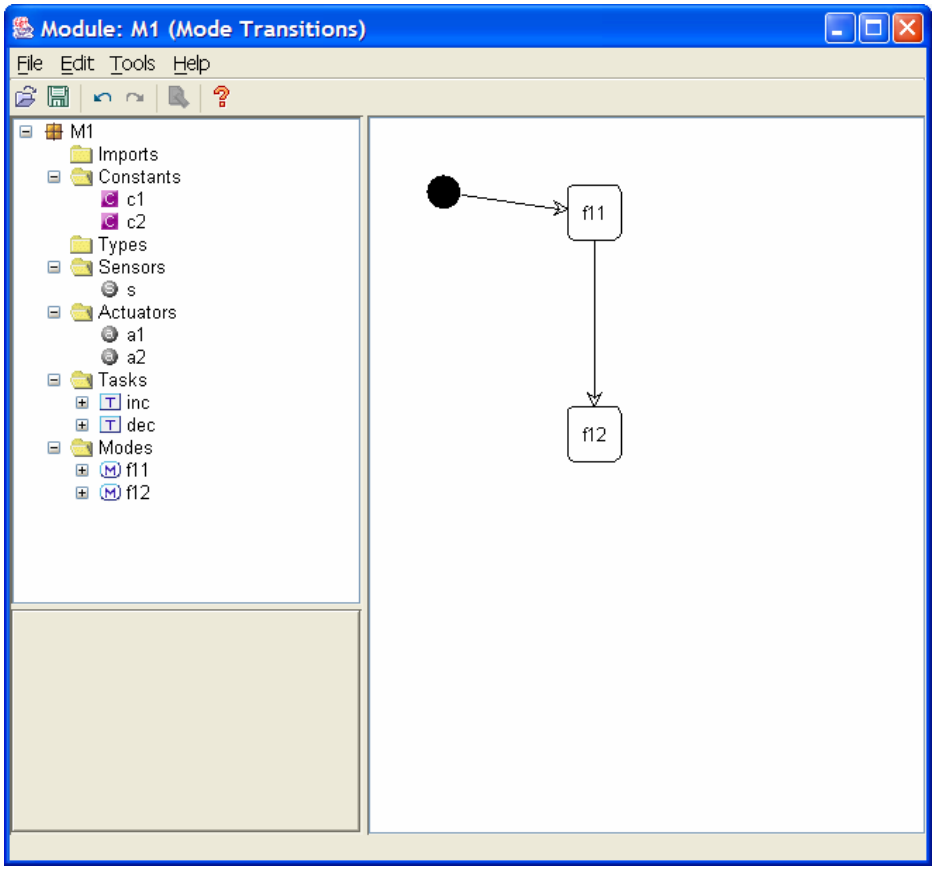
What is missing is the specification when a mode switch occurs. For that purpose we switch to the Mode Transition editor by selecting the module name *M1* or the *Modes* folder in the tree.



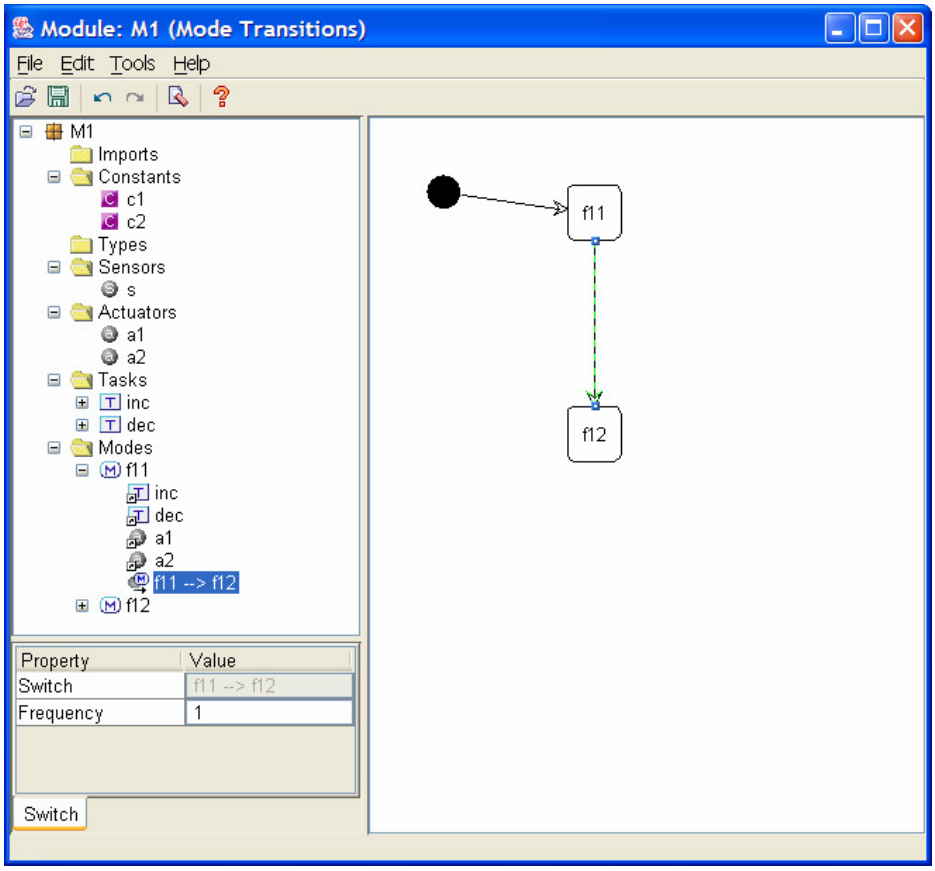
Then we add a mode transition from mode *f11* to mode *f12* by clicking in the *f11* box and dragging the connecting line to *f12*. Note that the mouse cursor has to be a hand when you are inside the mode box *f11*. The mouse cursor becomes a hand if you click the left mouse button inside the box that represents a mode, in this case mode *f11*, but a bit off the center area of the box. Only if the mouse cursor is a hand you can start connecting the two boxes that represent the modes *f11* and *f12*. Once you have started connecting the *f11* with *f12* make sure that before releasing the left mouse button you do that about in the center of the *f12* box.

If you want to move the boxes left-click inside the box at its center. Then the mouse cursor becomes a cross and you can move the particular mode represented by a box. The mouse pointer is a cross if you are in the center area of a box. Otherwise the mouse pointer is a hand for connecting modes.

After connecting the box *f11* with *f12* the Mode Transitions editor should look like this:

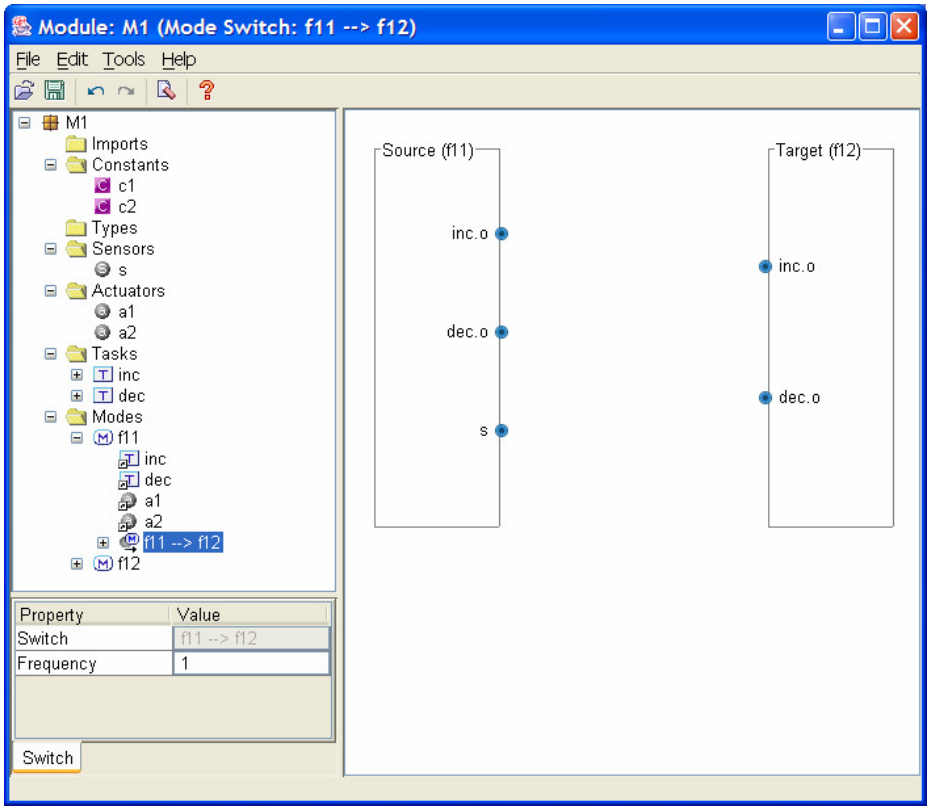


The arrow line that connects the boxes f11 and f12 represents a mode switch. Select the arrow line to see its properties in the inspector:

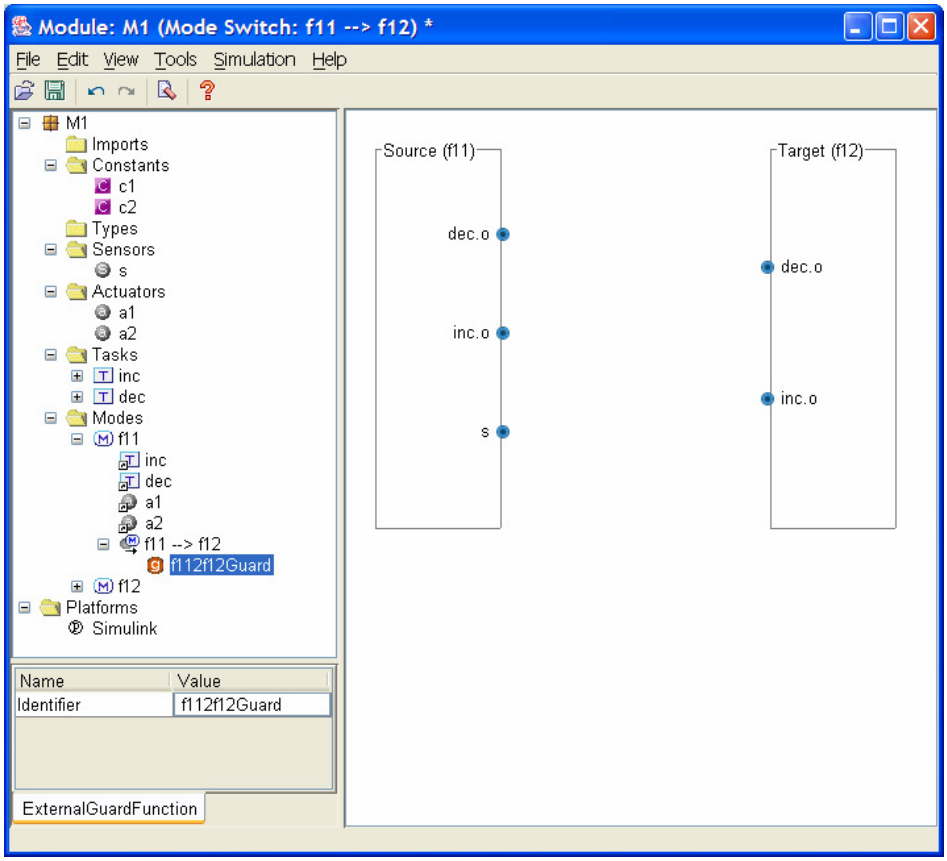


The mode switch is checked in mode f11, thus it is listed in the tree as an item labeled *f11→ f12* in the mode folder *f11*, which is a subfolder of the Modes folder. This item is selected in the tree in the above screen shot. The Frequency property is set to 1, which means that the mode switch is checked every 10 ms (mode period of mode *f11*) / 1 = 10 ms.

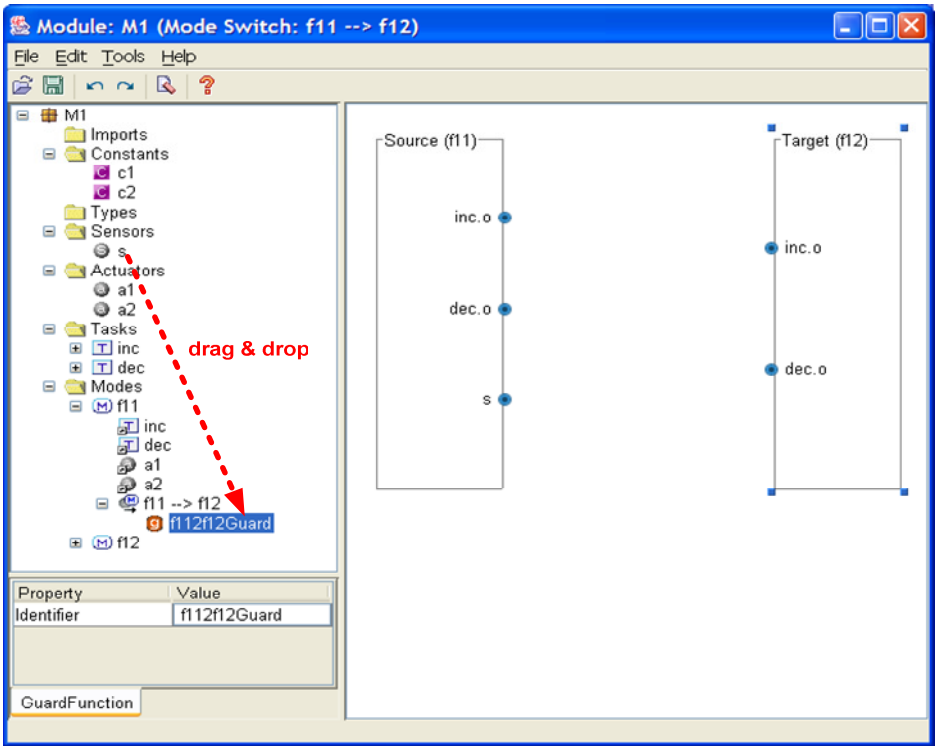
We now define that a mode switch from *f11* to *f12* should occur if the sensor *s* reads a value larger or equal to 1. Right-click on the mode switch item labeled *f11→ f12* in the tree and select the New Guard menu item in the pop-up menu. The window will look like the following screen shot, showing the Port Assignment editor in the right window pane (for the explanation of the purpose of that editor and how to use it see below).



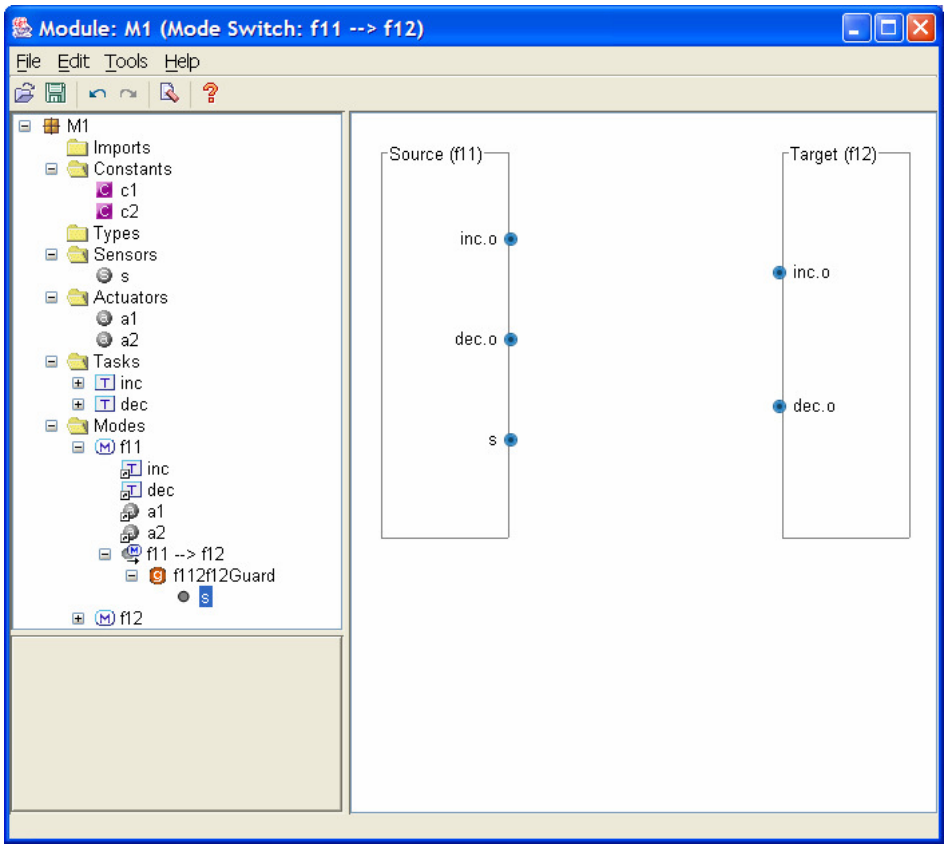
Click on the + next to the mode switch item labeled *f11→ f12* in the tree. This shows the guard function labeled *f112f12Guard* that we just added to the mode switch:



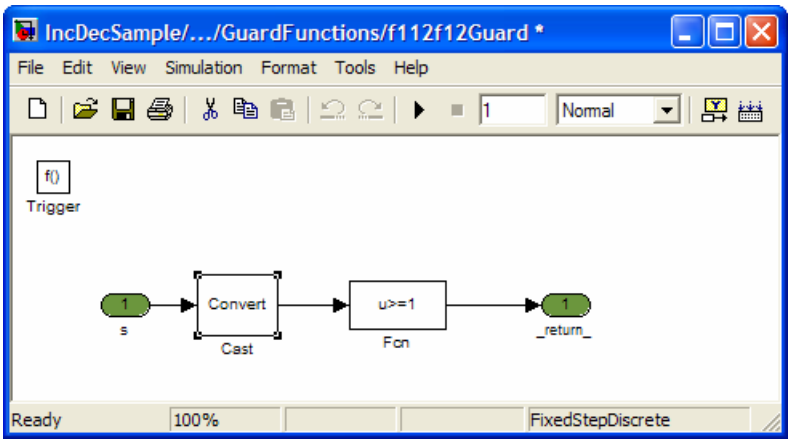
Analogous to task functions, guard functions can be specified in Simulink. Before doing that, we define which input the guard function needs: the guard function should cause a mode switch, if the sensor input has a certain value. Let us define that the mode remains *f11* if the sensor value *s* < 1 and that a mode switch should occur if the sensor value *s* >= 1. As a consequence, we need to access the sensor value for defining the guard function. Thus, we drag the sensor *s* from the tree's Sensors folder and drop it on the guard function *f112f12Guard* in the tree.



If you click on the + sign next to the guard function labeled *f112f12Guard* you see that the sensor *s* was added in the tree as input of the guard function:

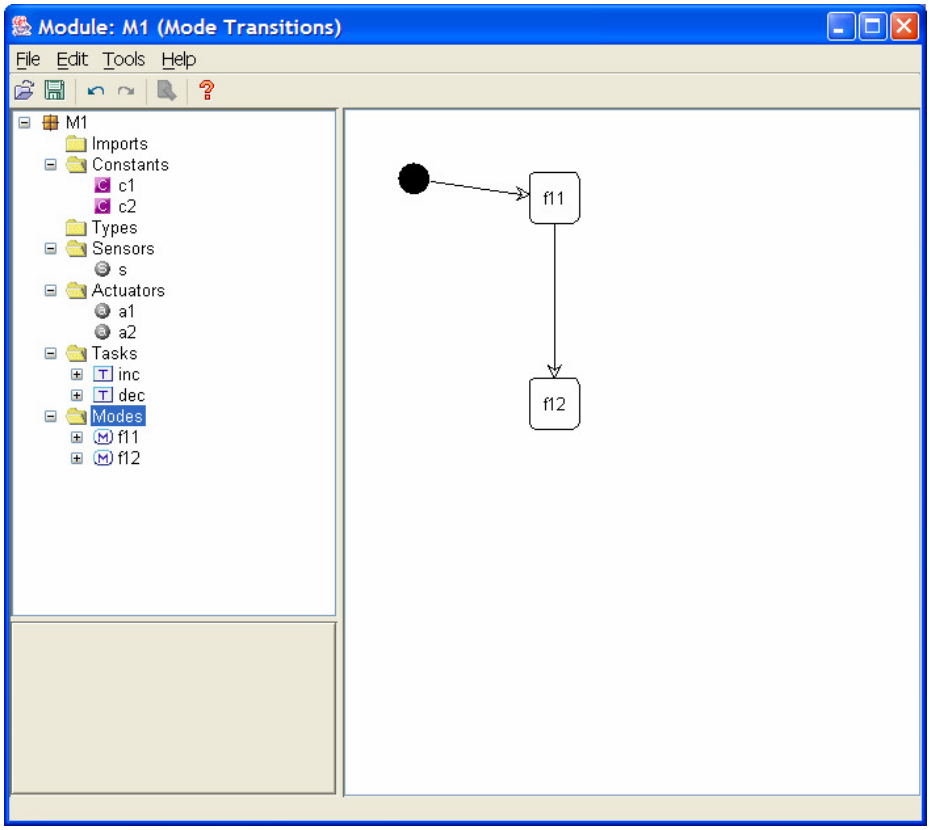


The guard function can now be modeled in Simulink® by simply double-clicking on the *f112f12Guard* item in the tree. This opens a Simulink® editor window. Add a Data Type Conversion block and a Fcn block from the Simulink® library and set the condition to u >= 1. The Fcn block returns 0 if the condition u (input) < 1 is true and returns 1 if the condition u >= 1 is true. The specification of the guard function should look like that:



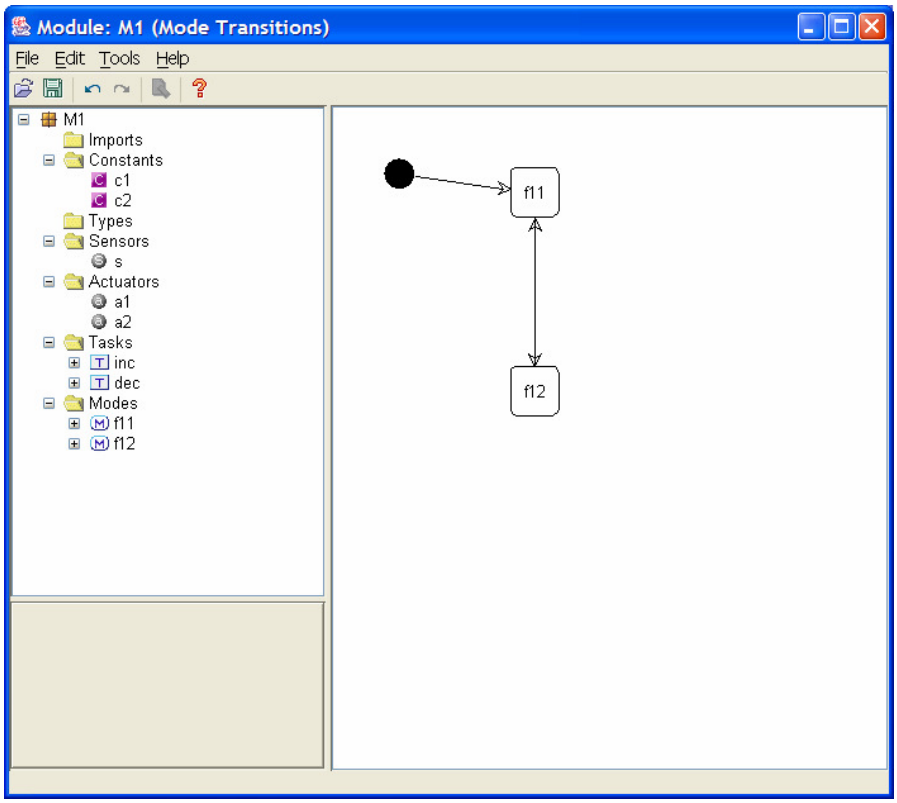
In an analogous way as the mode switch from *f11* to *f12* we define the mode switch from *f12* to *f11*. This mode switch should take place if the sensor *s* reads a value smaller than 1 (s < 1).

For that purpose we switch to the Mode Transition editor by selecting the module name *M1* or the *Modes* folder in the tree.

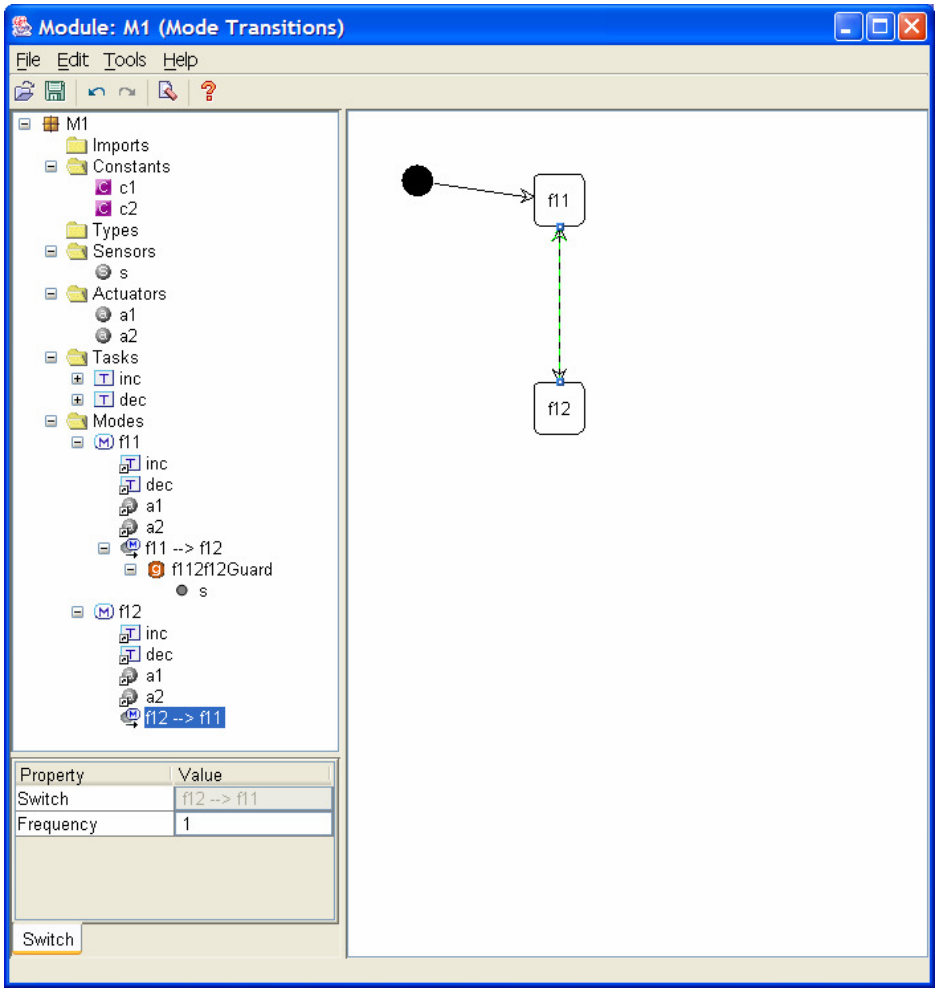


Then we add a mode transition from mode *f12* to mode *f11* by clicking in the *f12* box and dragging the connecting line to *f11*. Remember that the mouse cursor has to be a hand when you are inside the mode box *f12*. Once you have started connecting the *f12* box with the *f11* box make sure that before releasing the left mouse button you do that about in the center of the *f11* box.

After connecting the box *f11* with *f12* the Mode Transitions editor should have arrows on both ends of the line connecting modes *f11* and *f12* and thus should look like this:

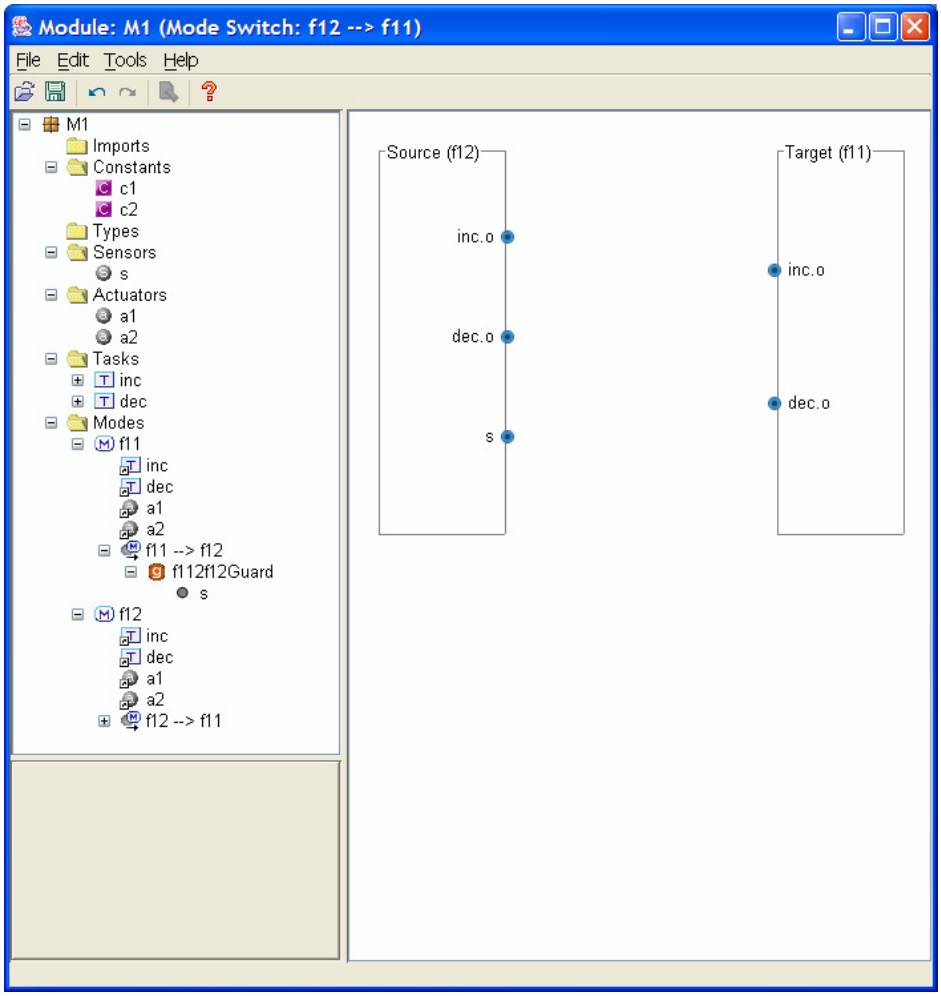


When you select the arrow line (one left mouse click on the line) the corresponding item is selected in the tree and you see the properties of the mode switch in the one direction, that is from f11 to f12, in the inspector. If you left-click again, you see the properties of the switch from f12 to f11 in the inspector. Each further left-click on the connecting line with the two arrows toggles the selected switch. Make sure you have selected the switch from f12 to f11:

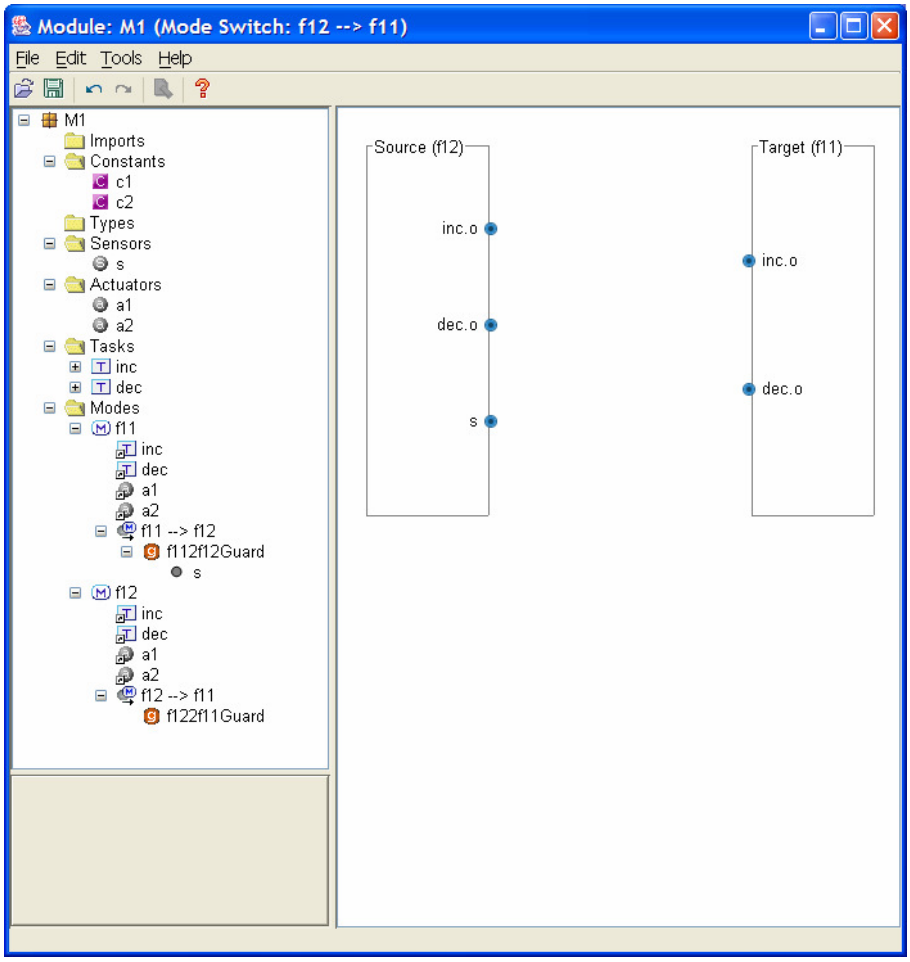


The mode switch is checked in mode f12, thus it is listed in the tree as an item labeled *f12* → *f11* in the mode folder *f12*, which is another subfolder of the Modes folder. This item is selected in the tree in the above screen shot. The Frequency property is set to 1, which means that the mode switch is checked every 10 ms (mode period of mode *f12*) / 1 = 10 ms.

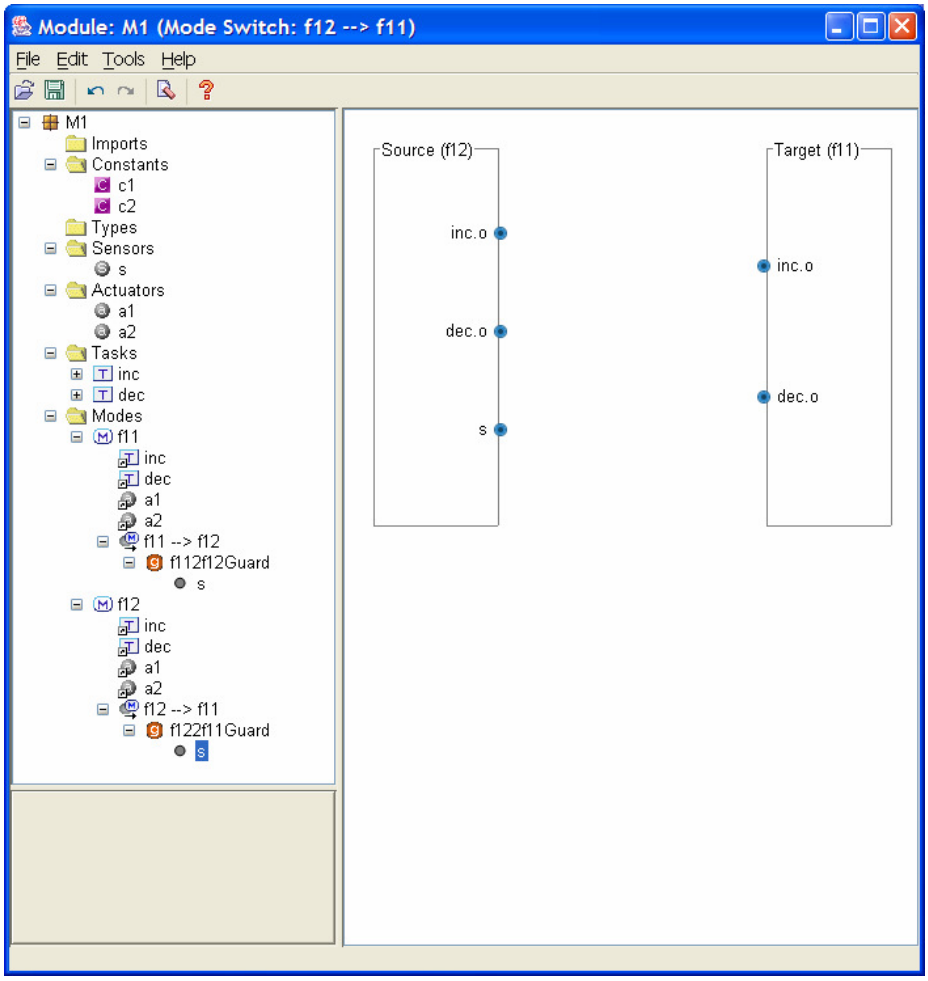
We now define that a mode switch from *f12* to *f11* should occur if the sensor *s* reads a value smaller than 1. Right-click on the mode switch item labeled *f12* → *f11* in the tree and select the New Guard menu item in the pop-up menu. The window will look like the following screen shot, showing the Port Assignment editor in the right window pane (for the explanation of the purpose of that editor and how to use it see below).



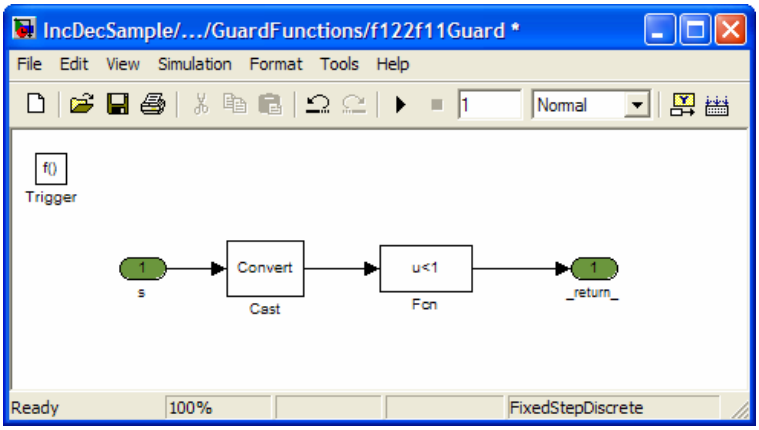
Click on the + next to the mode switch item labeled *f12 → f11* in the tree. This shows the guard function labeled *f122f11Guard* that we just added to the mode switch:



Drag the sensor *s* from the tree's Sensors folder and drop it on the guard function *f122f11Guard* in the tree. Click on the + sign next to the guard function labeled *f122f11Guard* and you see that the sensor *s* was added in the tree as input of the guard function:

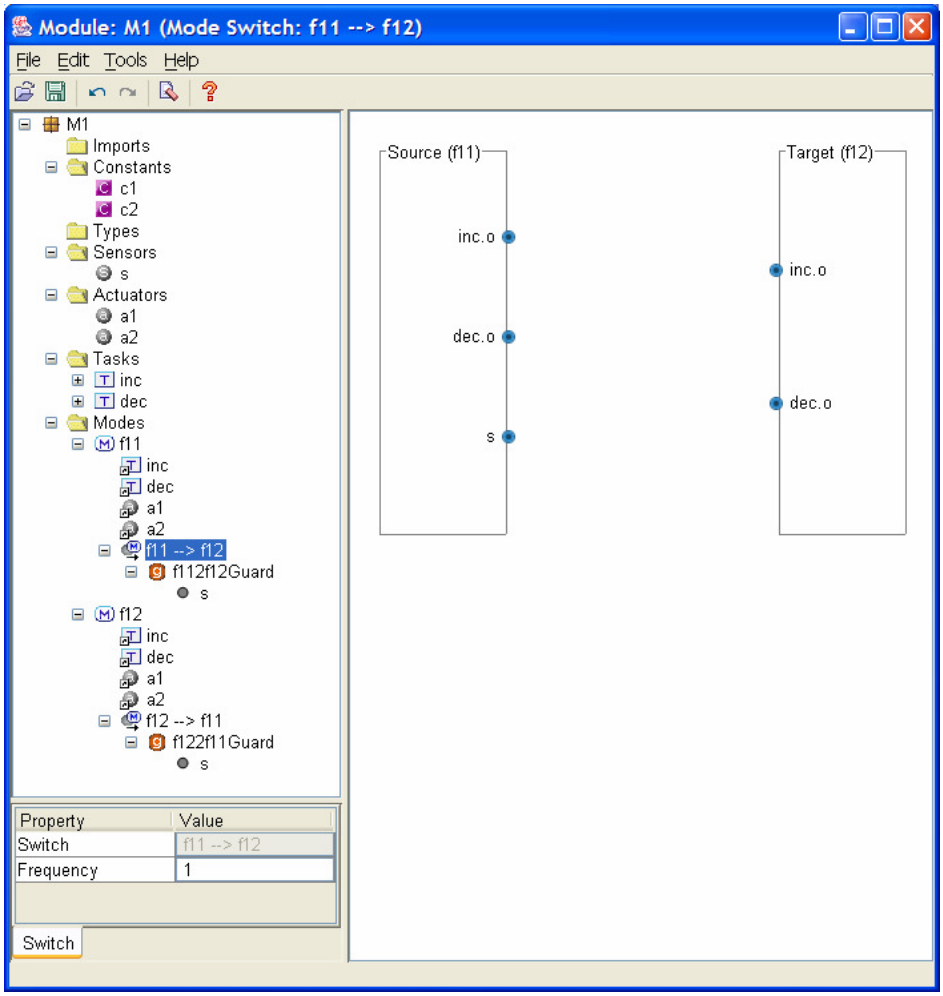


The guard function can again be modeled in Simulink® by simply double-clicking on the *f122f11Guard* item in the tree. This opens a Simulink® editor window. Add a Data Type Conversion block and a Fcn block from the Simulink® library and set the condition to u < 1. The Fcn block returns 1 if the condition u (input) < 1 holds true and returns 0 if the condition u >= 1 holds true. The specification of the guard function should look like that:

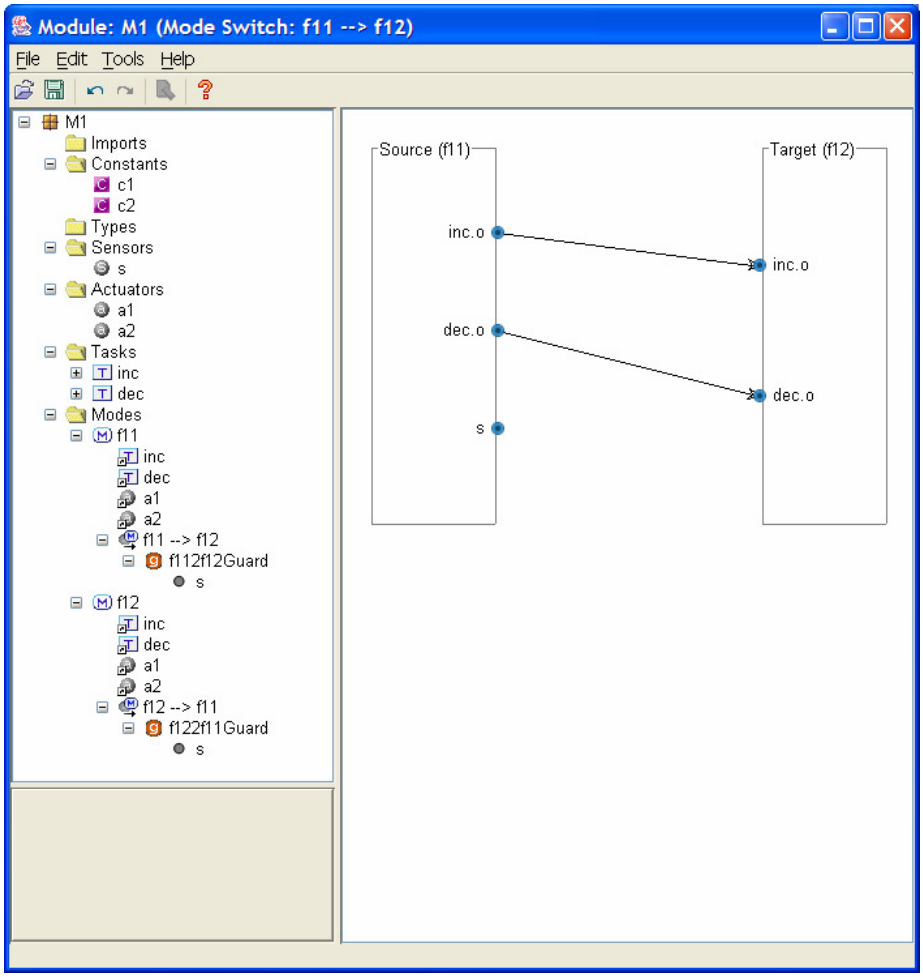


Finally, we specify by means of the Port Assignment editor how output ports of a mode's tasks are initialized after a mode switch. Let us do that first for the mode

switch from *f11* to *f12*. For this purpose select the mode switch item labeled *f11*→ *f12* in the tree:

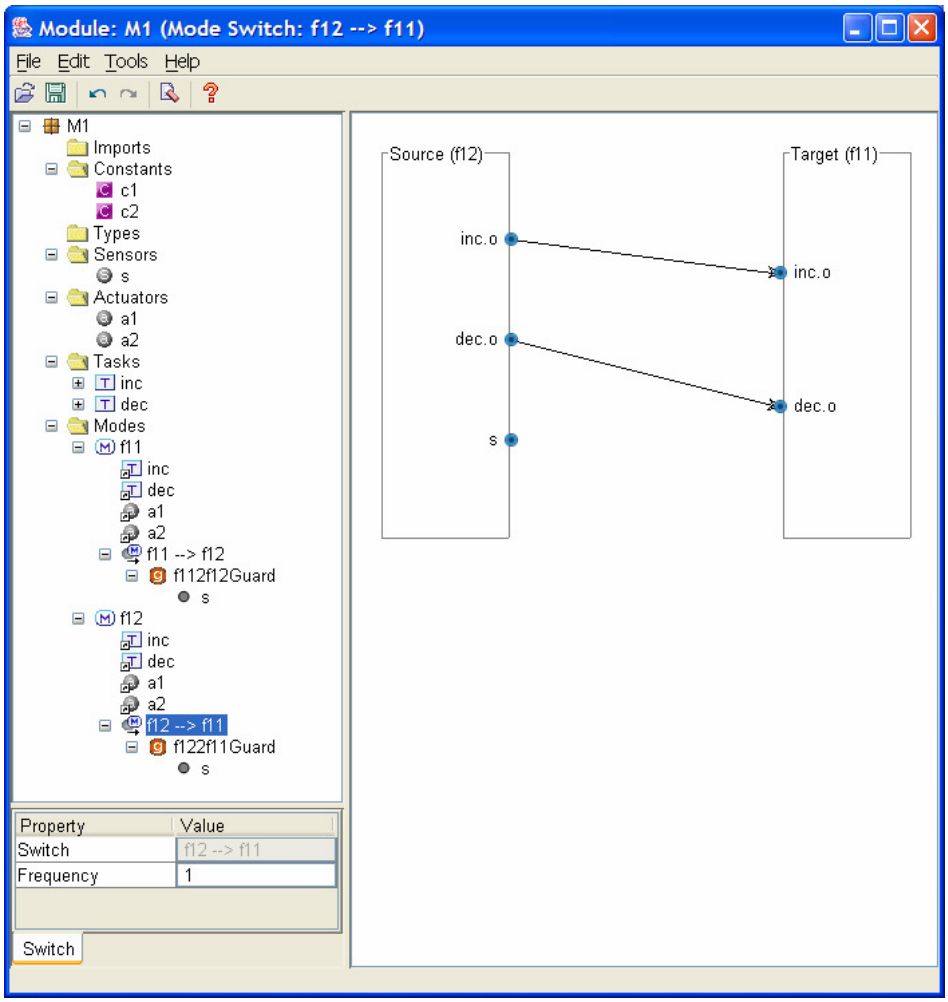


We connect the output port of task *inc* in *f11* with the output port of task *inc* in *f12*. And we also connect the output port of task *dec* in *f11* with the output port of task *dec* in *f12*:



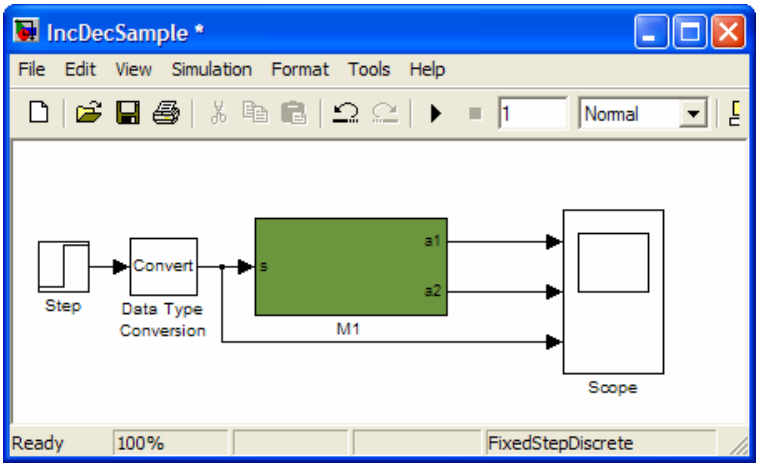
This corresponds to the statement { inc.o := inc.o; dec.o := dec.o; } after the mode switch condition in the textual version of the TDL program. For the sake of clarity this was not shown in the textual version of the modules in the previous chapter.

In an analogous way we define how output ports of the tasks of mode *f12* are initialized after a mode switch from *f11*:

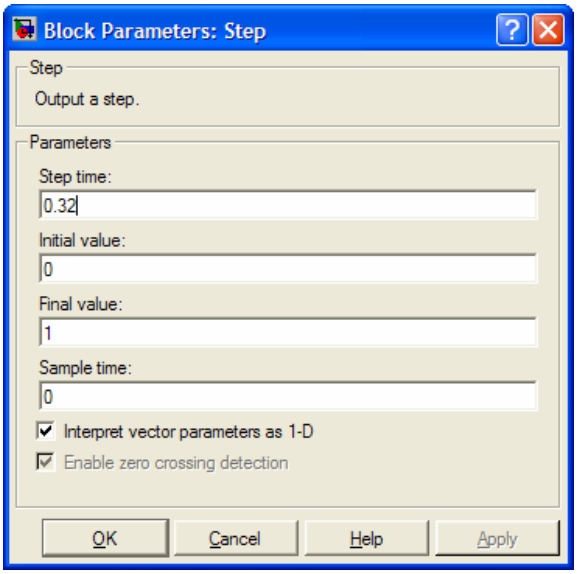


Test module M1

In the top-level subsystem add a Step block together with a Data Type Conversion block as sensor input of *M1* and a Scope block with three input ports to watch the output behavior of the two actuators *a1* and *a2* of *M1* and of the generated sensor input:



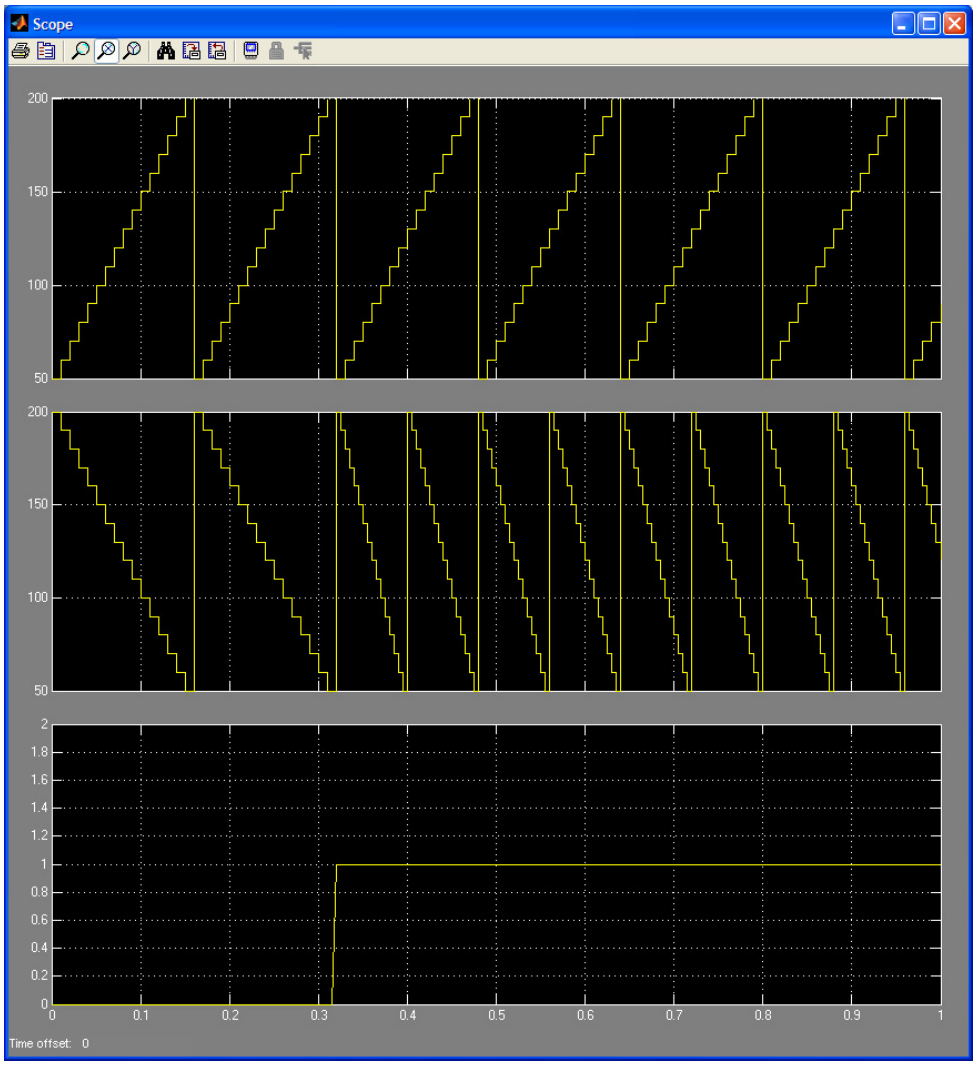
Set the simulation stop time to 1.0, that is 1 second, as shown in the window above. Define the Step block properties as shown in the following dialog:



The Step block generates an ouput value of 0 for the first 319 milliseconds. Then it generates an output value 1. Thus the module *M1* is in mode *f11* for the first 319 milliseconds and then in mode *f12*.

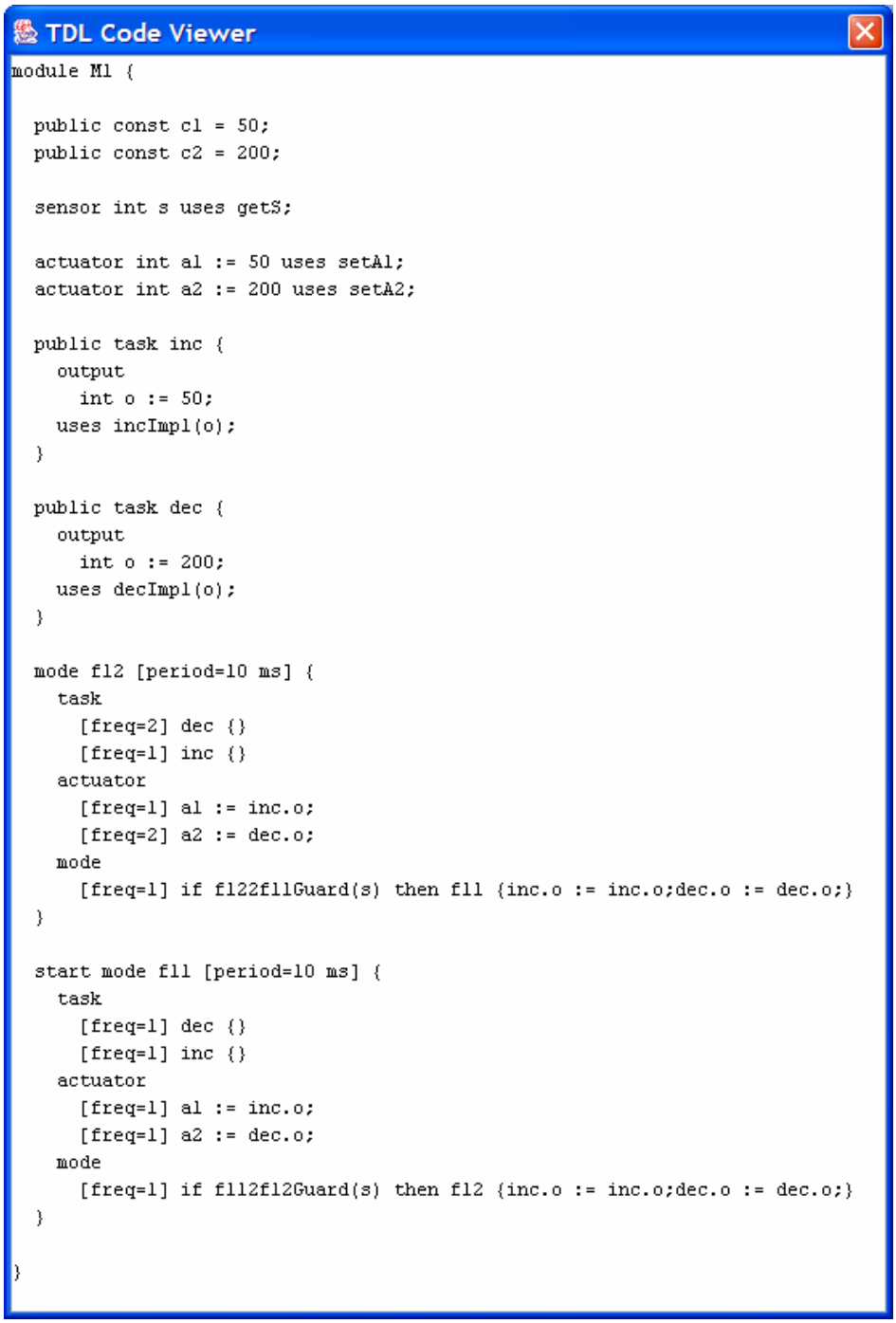
Simulation. Start the simulation either by pressing the Start simulation icon in the tool bar of the Simulink® editor or choose the Start menu item in the Tools menu of the Simulink® editor (Tools>Start). This causes the TDL:VisualCreator tool to invoke the TDL:Compiler and to generate internally used Simulink® blocks that deliver the TDL semantics when the overall model is simulated.

After the simulation has finished, the Scope block should show the following output:



TDL source code of module M1

Choose the File>View TDL Code menu item in the TDL:VisualCreator tool. This opens a read-only text viewer that shows the TDL program text of module *M1*:

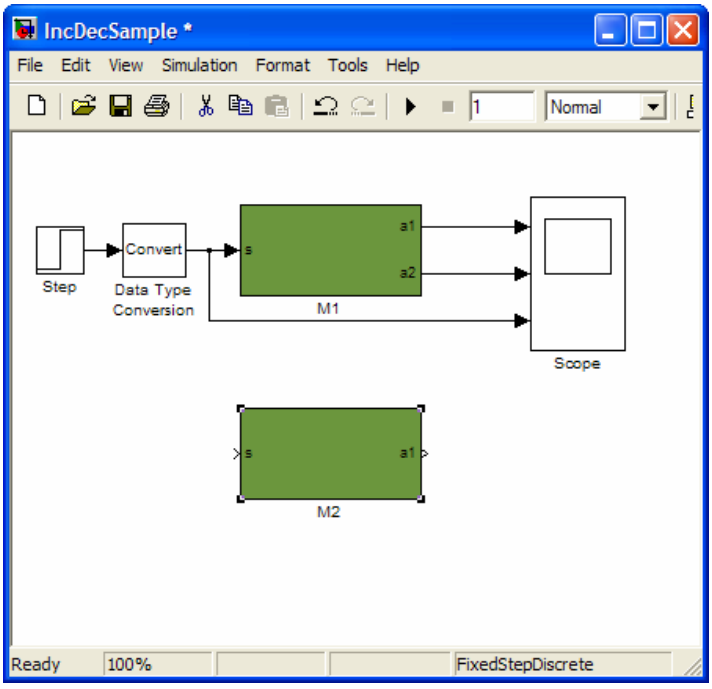


Define module *M2*

Module *M2* exemplifies how a module imports another module. In this case *M2* imports *M1* to access the output ports of M1's tasks *inc* and *dec*. Module *M2*'s task *sum* simply adds the outputs of *M1*'s *inc* and *dec* tasks.

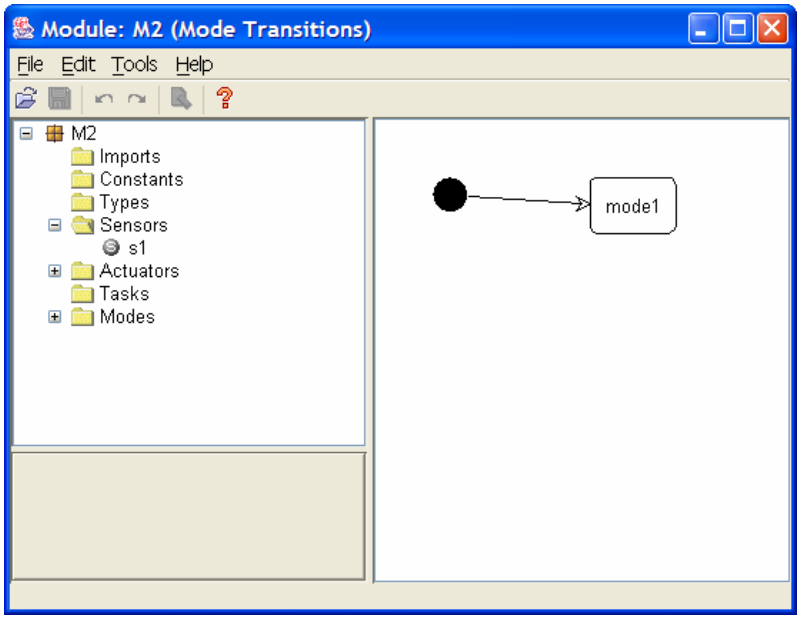
Add module *M2*

Add another TDL module from the Simulink® Library Browser and rename it to *M2*:

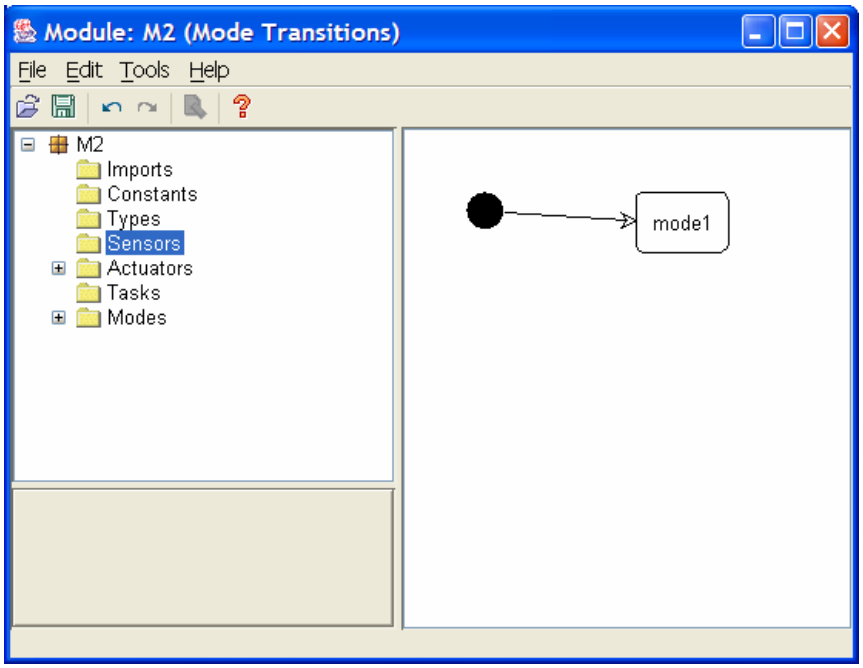


Remove sensor

As module *M2* only uses the outputs of *M1*'s tasks we do not need a sensor. Double-click on the *M2* module block to open the TDL:VisualCreator tool and click on the + symbol left to the *Sensors* folder in the tree:

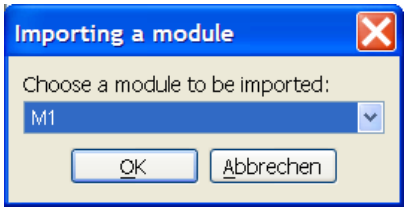


Select the *s1* item and choose the Edit>Delete menu item to remove the sensor. Alternatively you can right-click on the *s1* item and choose Delete from the pop-up menu. After that operation the Sensors folder is empty:

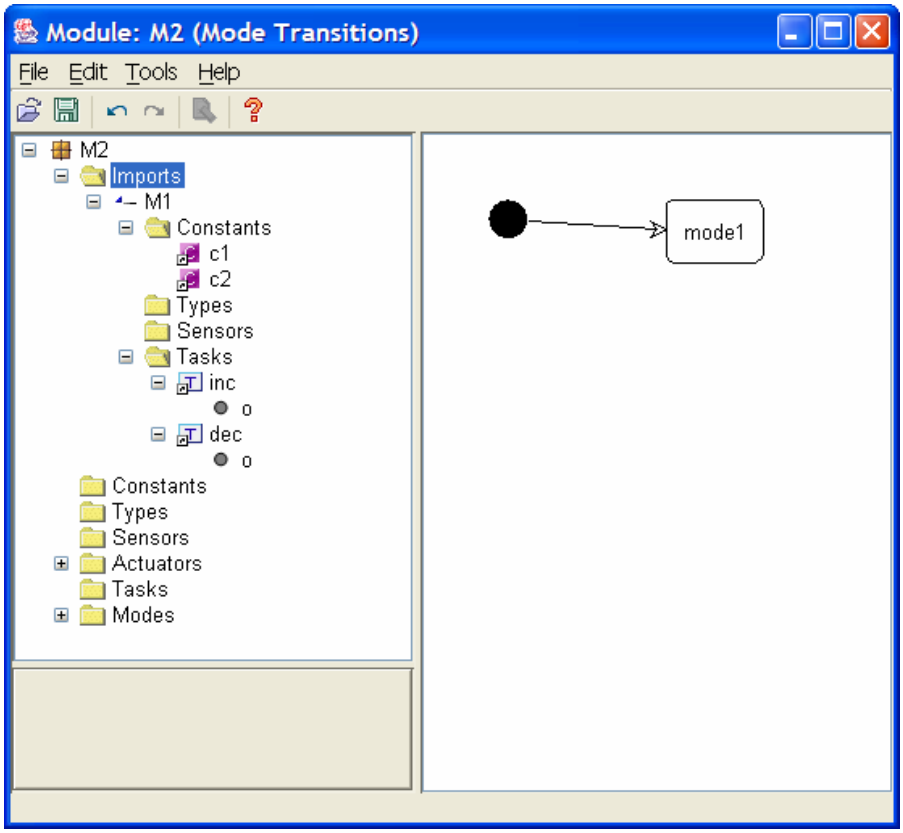


Import module *M1*

Choose the File>Import menu item or right-click on the *Imports* folder in the tree. This opens the following dialog:

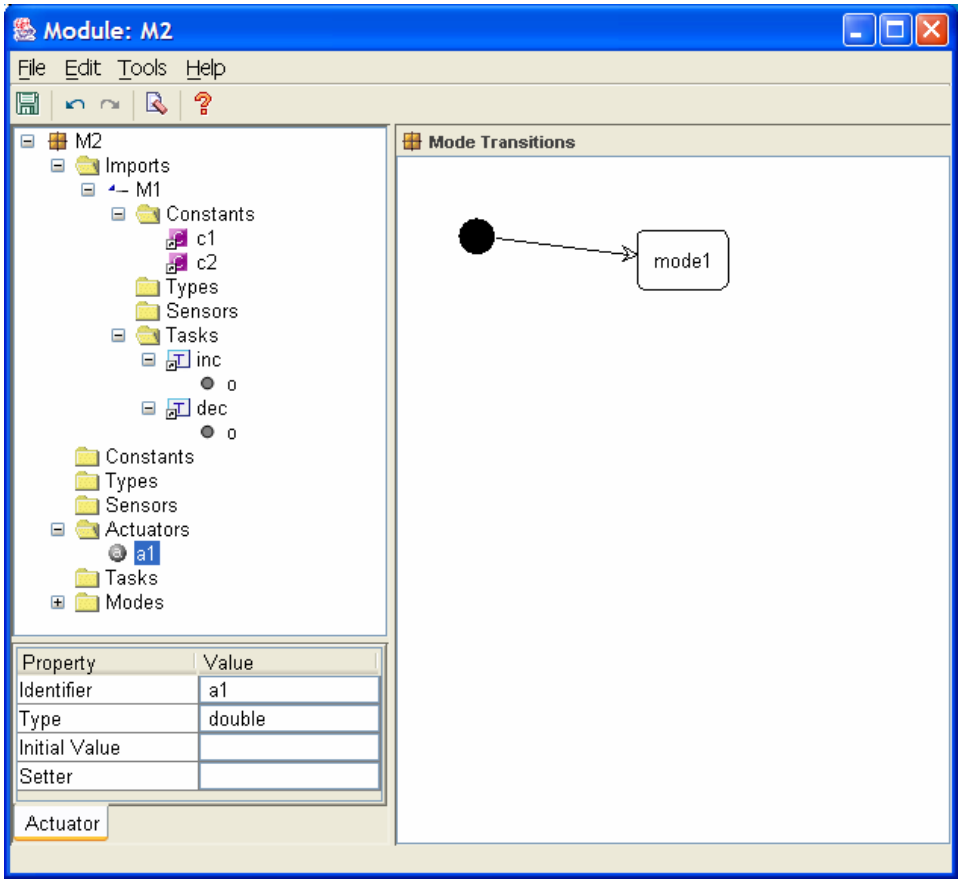


Press the OK button to import *M1* into *M2*. Click on the + symbols left to the Imports folder and left to the items listed in this folder. All public members of module *M1* are listed:



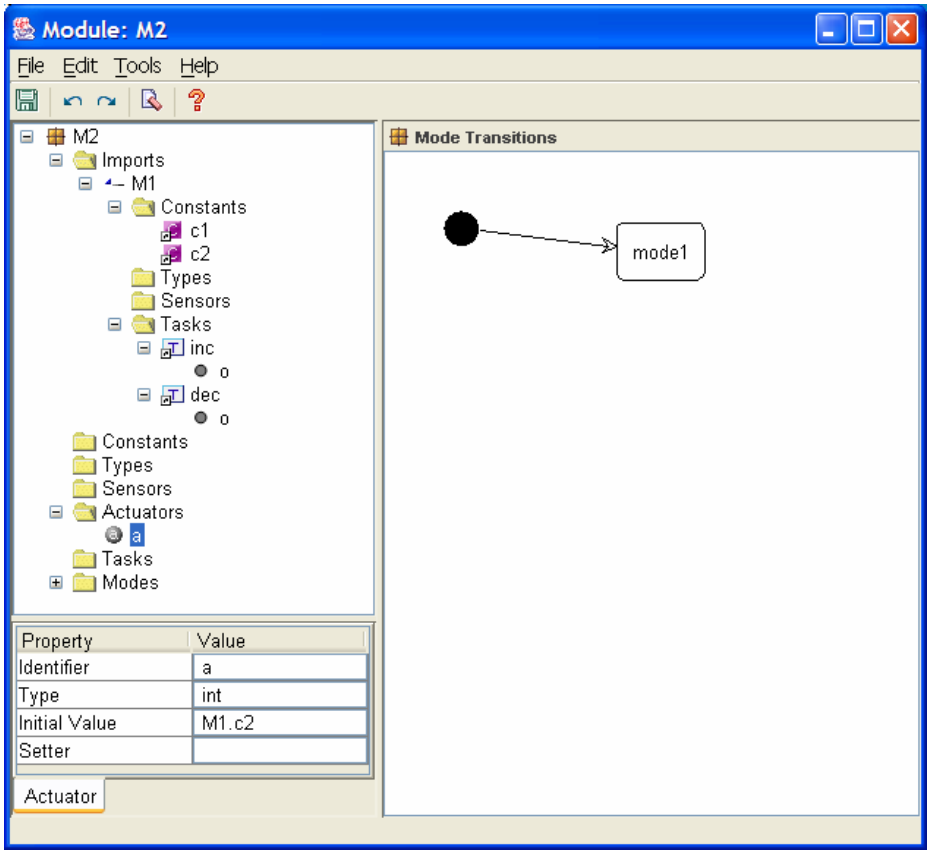
Edit actuator *a*

Each new module has already one actuator. In order to rename this actuator open the *Actuators* folder (click on the + symbol left to the *Actuators* folder in the tree pane):



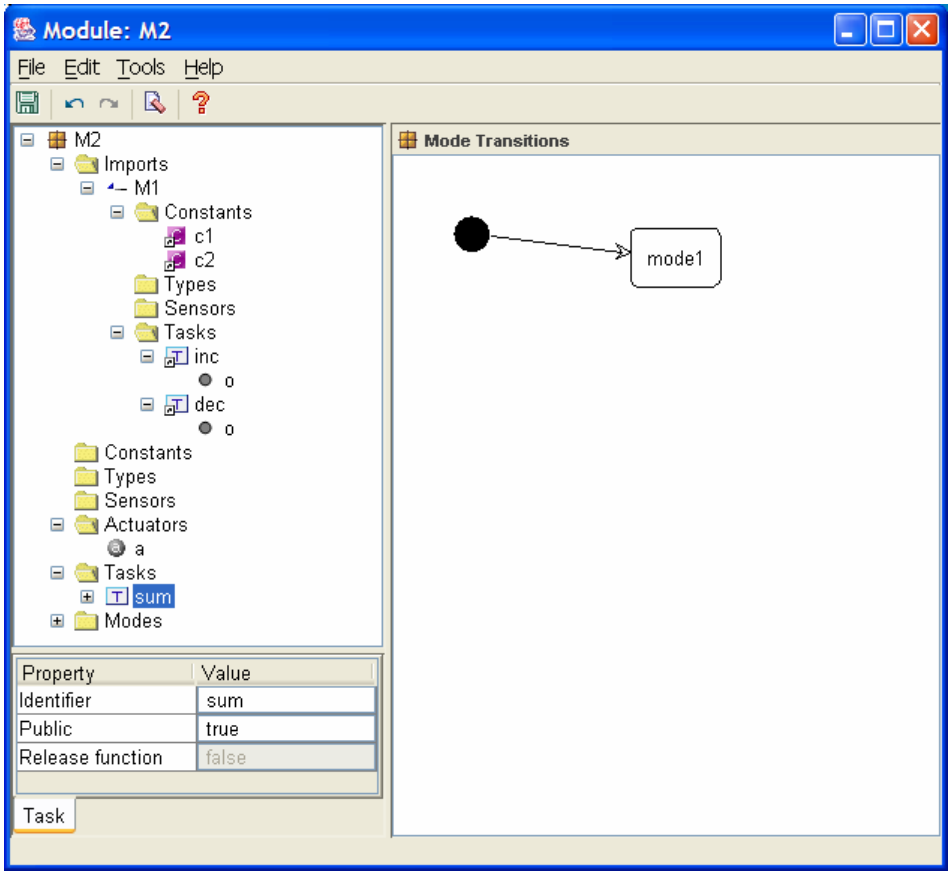
Either choose the Edit>Rename menu item or right-click on the listed *a1* item and choose Rename from the pop-up menu. Rename the actuator from *a1* to *a*. As with the sensors and all other TDL constructs you can also click twice (no double-click speed!) on the *a1* item in the tree to edit the name directly or you can change the Identifier in the inspector.

Also change the actuator Type property to integer (int), and the Initial Value property to M1.c2:

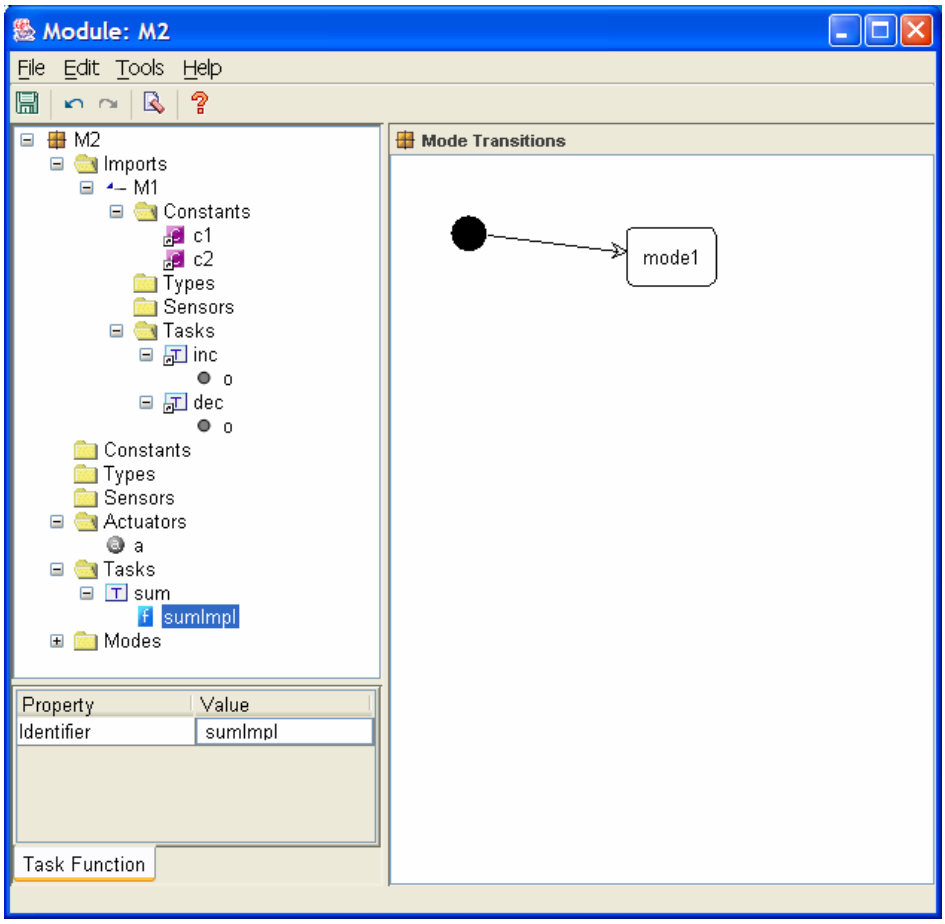


Edit task *sum*

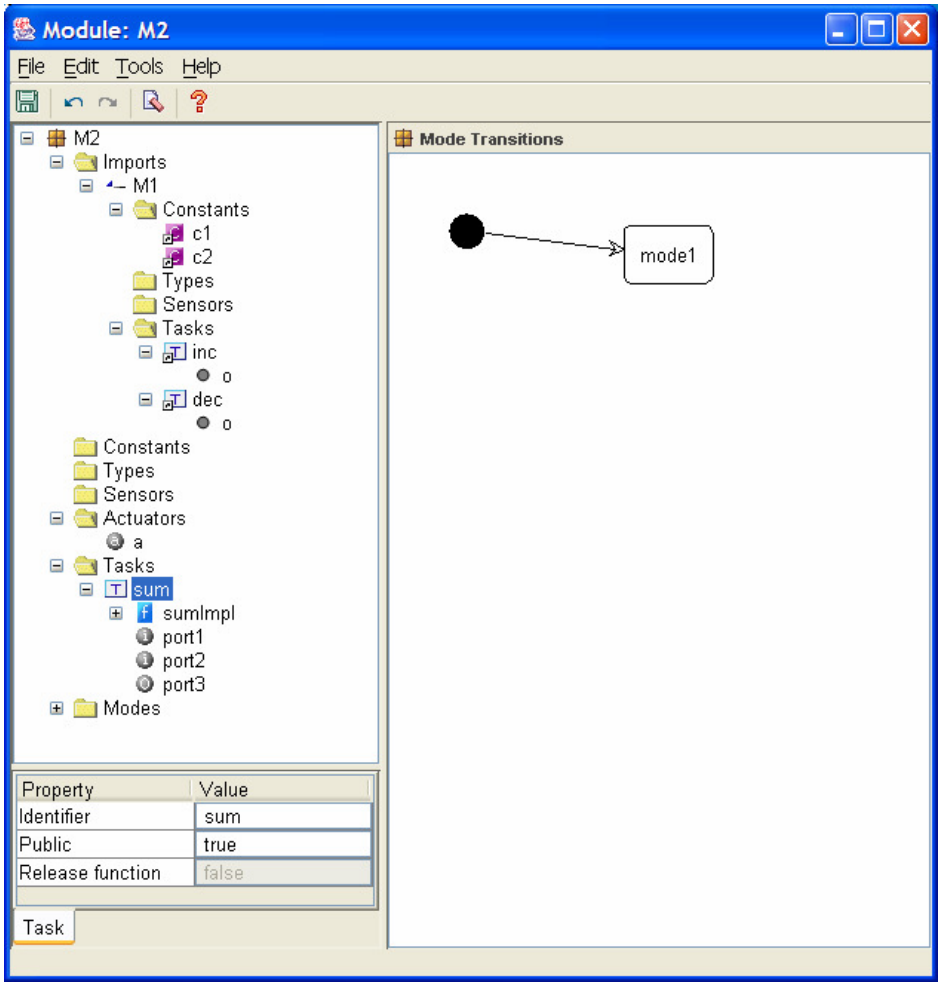
From the File>New submenu choose Task or right-click on the *Tasks* folder in the tree and choose the New Task item in the pop-up menu. Rename the task from *t1* to *sum*, and set the Public property to true.



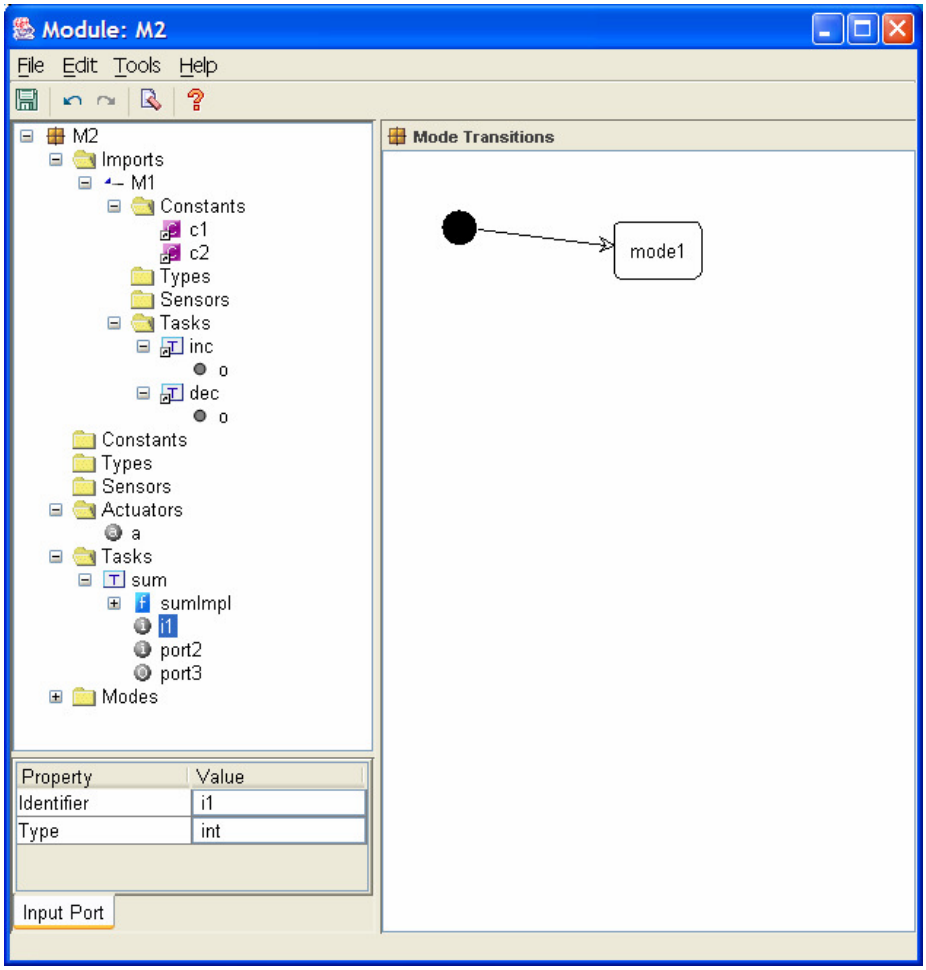
Click the + left to task *sum* and rename the task function from *t1Impl* to *sumImpl*.



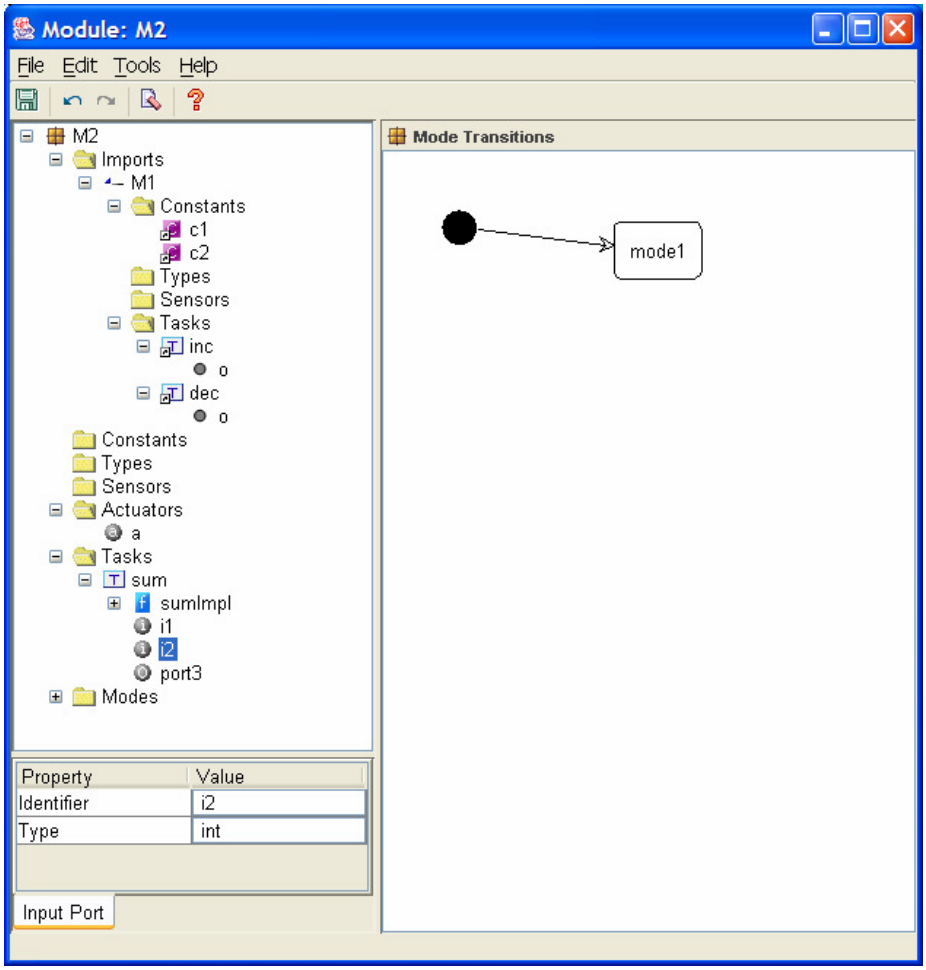
The task needs two input ports and one output port. Add these ports to task *sum* by right-clicking on the *sum* item in the tree and selecting twice the New input port and then once the New output port menu item in the pop-up menu. After adding the two input ports and one output port the tree should look as follows:



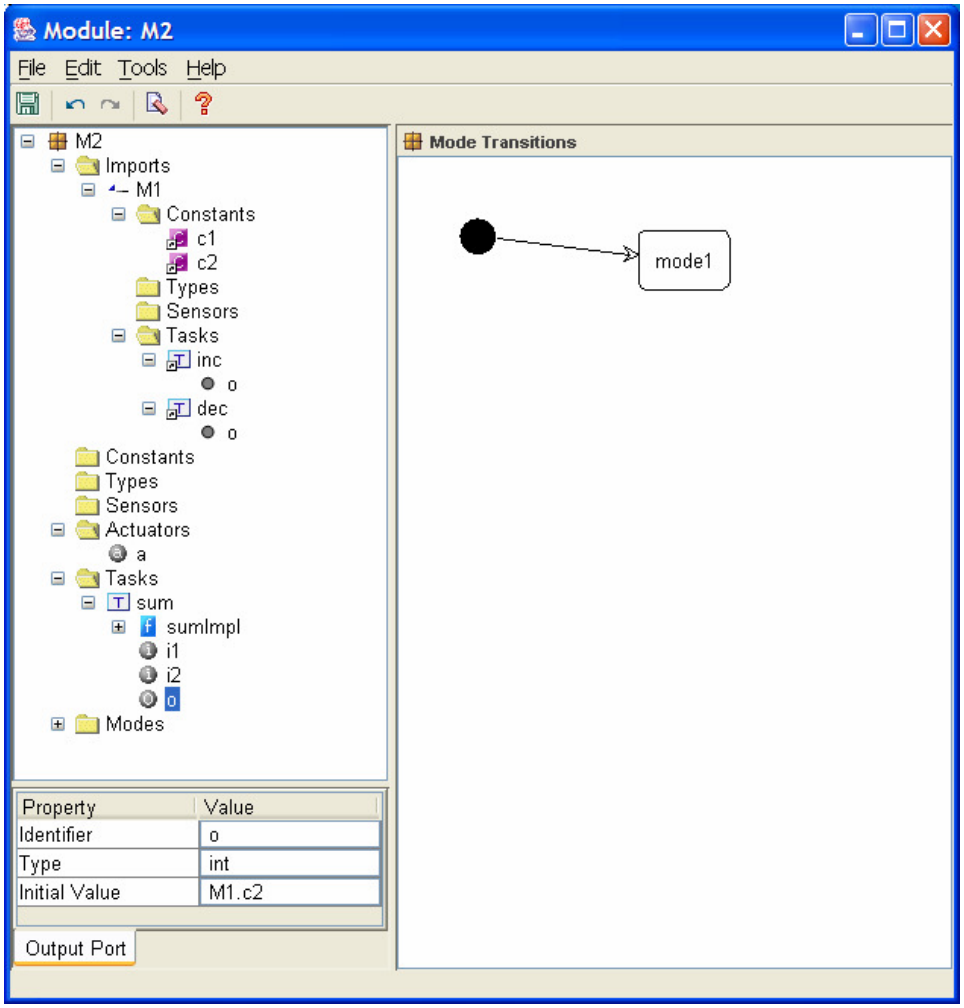
Rename *port1* to *i1* and set its type to int:



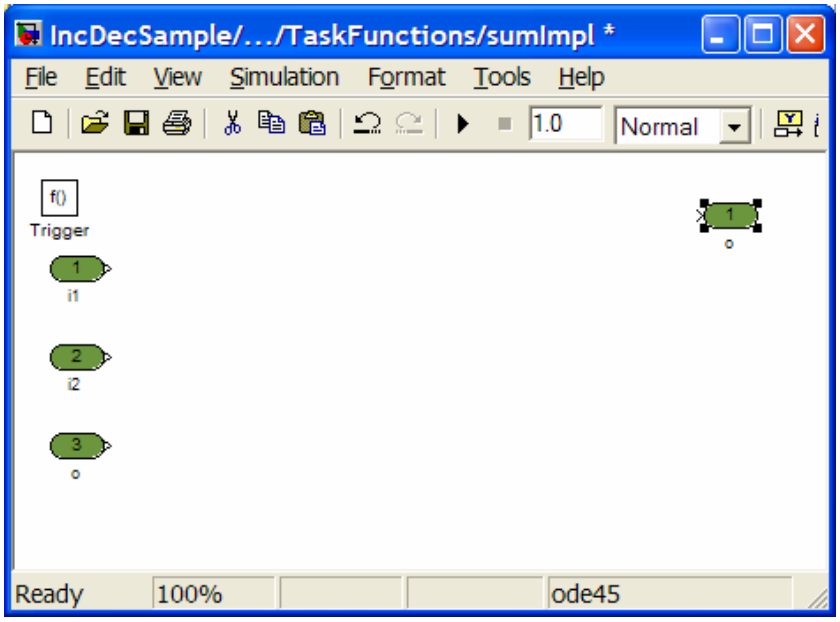
Rename *port2* to *i2* and set its type to int:



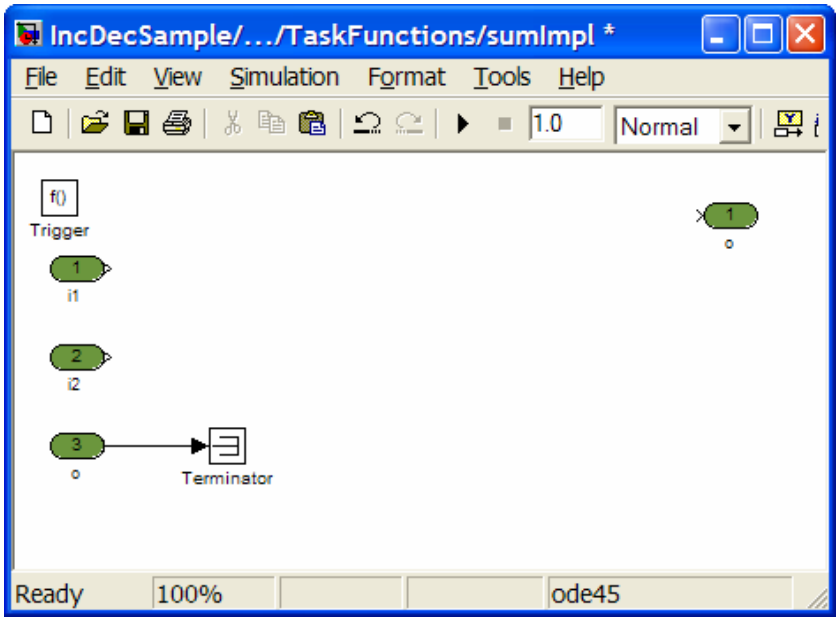
Rename *port3* to *o*, set its type to int, and define the Initial Value as *M1.c2*:



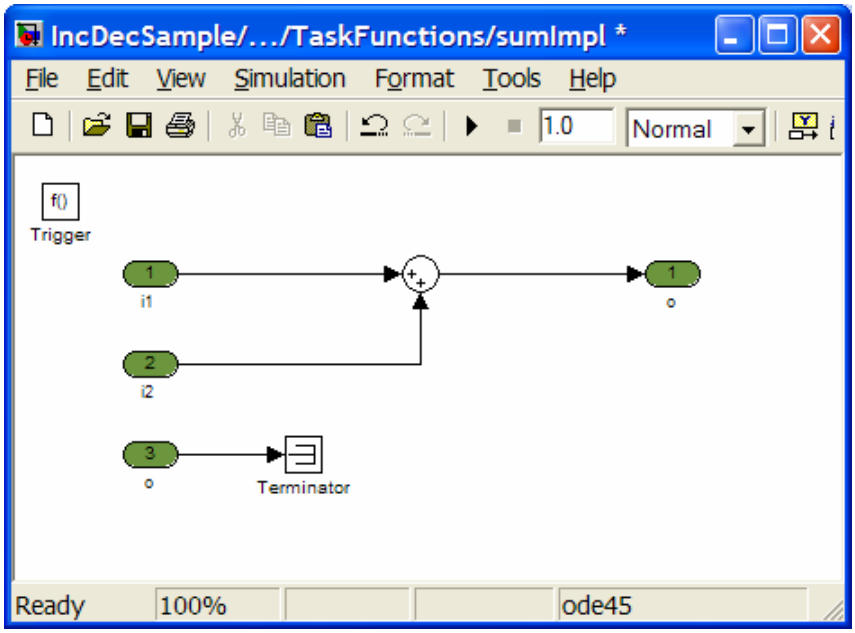
The task function can be modeled in Simulink® by simply double-clicking on the *sumImpl* item in the tree. This opens a Simulink® editor window:



The outport *o* corresponds to the output port of the task. The inport o provides the value of *o* as calculated in the previous task invocation. As we do not need this value in this case we connect it to a terminator block:

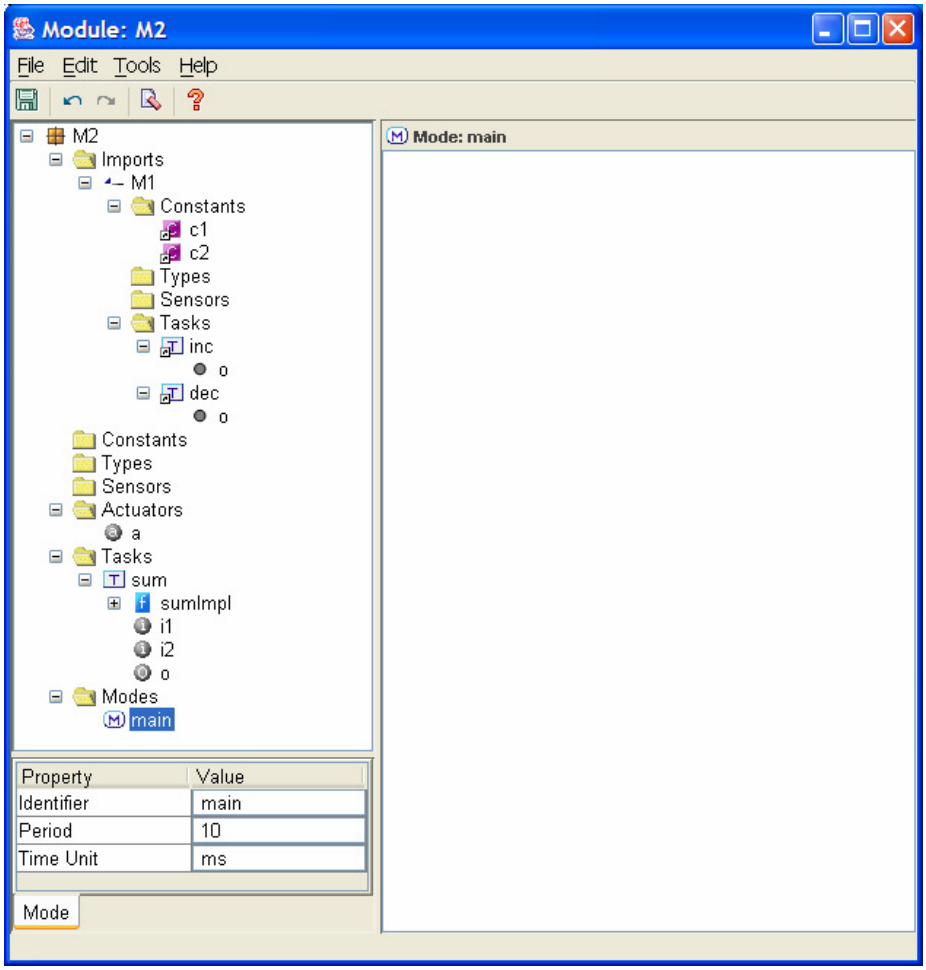


We model the straight-forward functionality of task *sum*, that is, calculating the sum of its input ports *i1* and *i2*, by means of the standard Simulink® block Sum:



Edit mode *main*

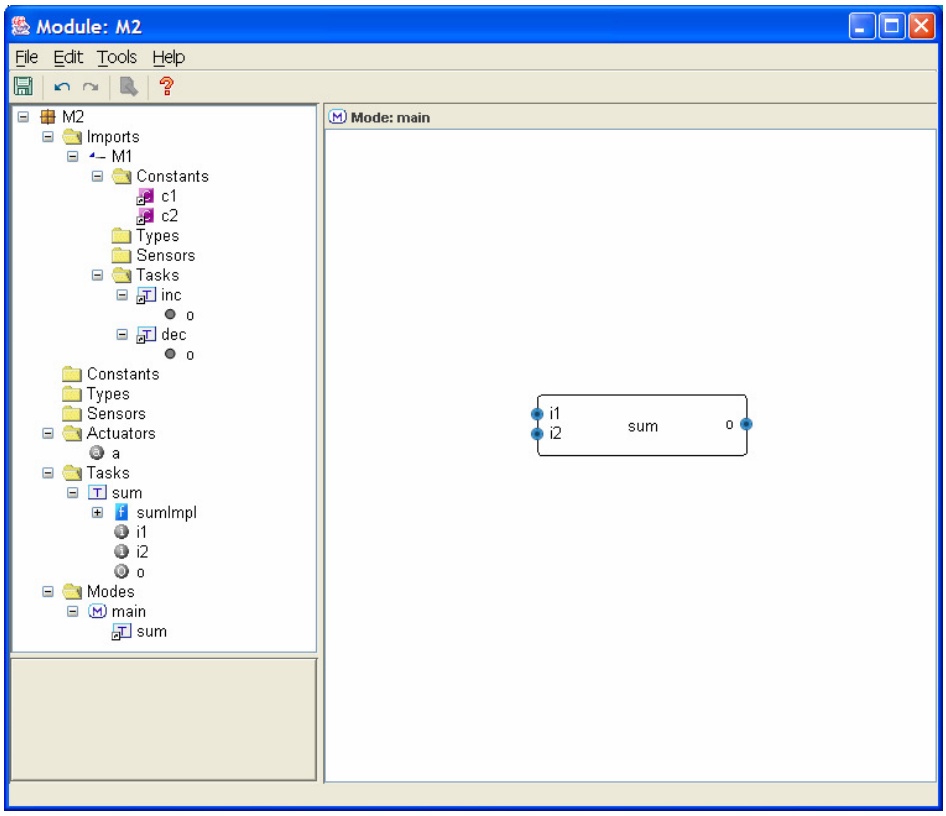
Each new module has already one mode called *mode1*. Rename this mode to *main*, and set the mode period of mode *main* to 10 ms in the inspector:



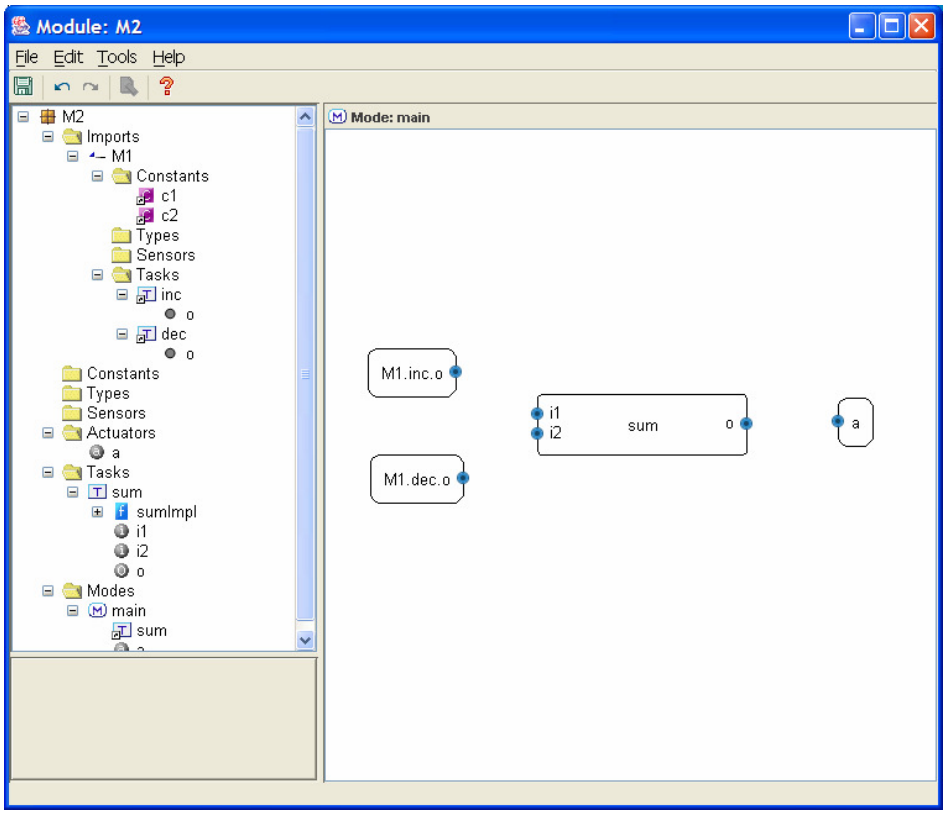
Task invocation and actuator update. To assign task *sum* to mode *main* we activate the Mode editor by selecting the *main* item in the tree pane.

Remember that you can switch back to the Mode Transition editor by either selecting the *Modes* folder or the module name (*M2* in this case) in the tree.

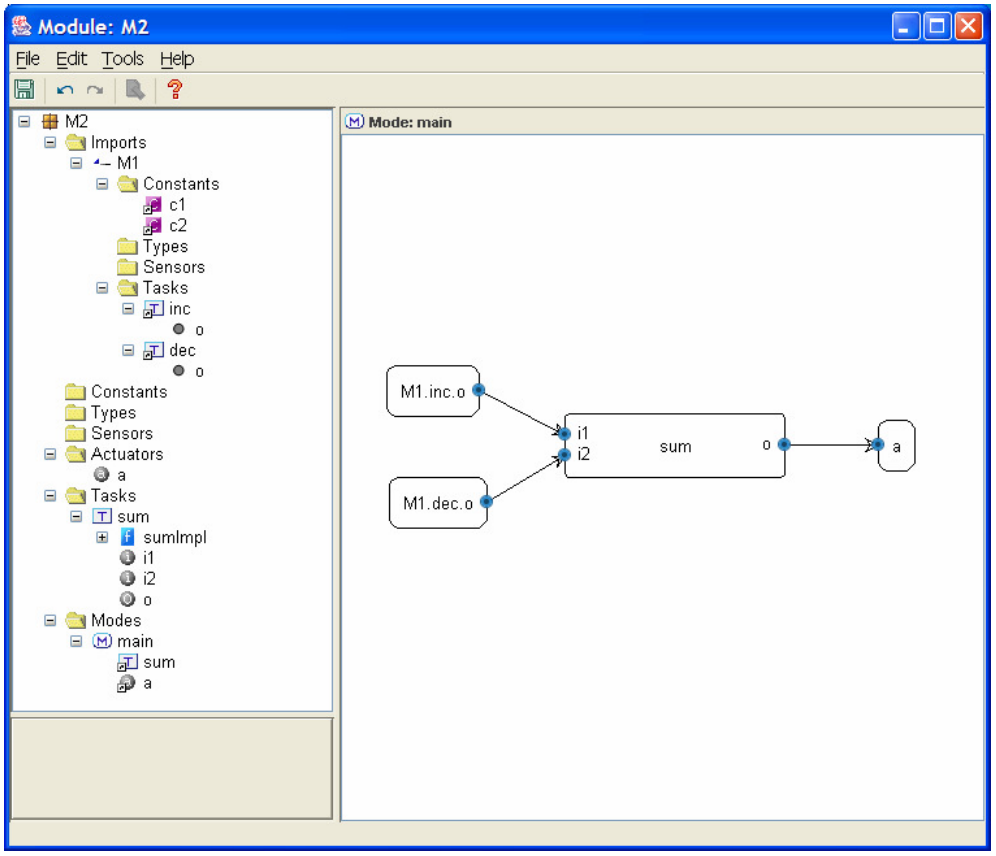
For now we keep the Mode editor active to edit mode *main*. Drag the item in the tree that represents task *sum* and drop it on the right Mode editor pane. The task *sum* is displayed as box with its name as well as its input and output ports. If you select the task instance you see that the Frequency property is set to 1. As the mode period of *main* is 10 ms this means that *sum* has a Logical Execution Time (LET) of 10 ms in mode *main*:



Next we drag *M1.inc.o* and *M1.dec.o* as well as actuator *a* from the tree and drop them on the Mode editor:

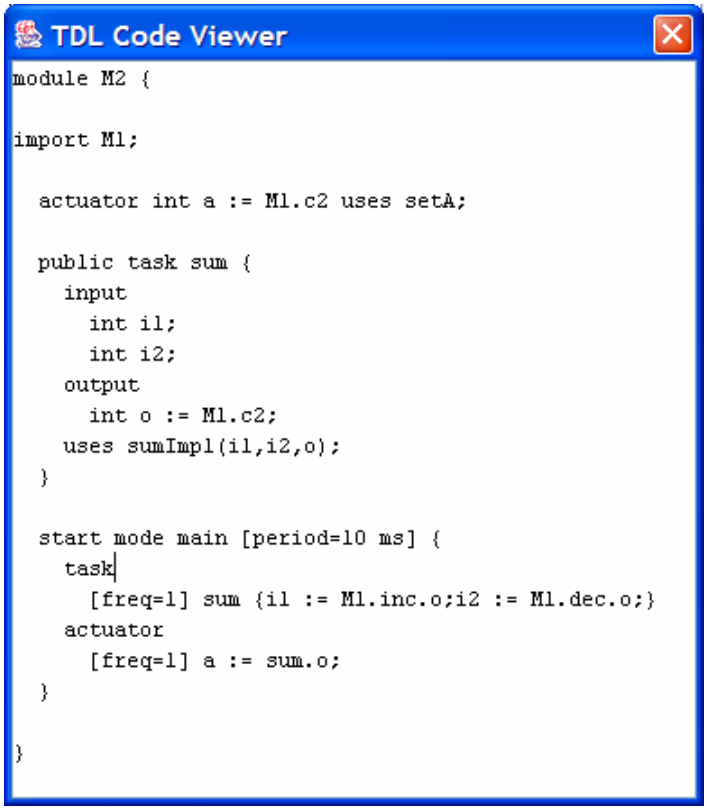


Now we can connect *M1.inc.o* to *i1*, *M1.dec.o* to *i2* and *o* to *a*:



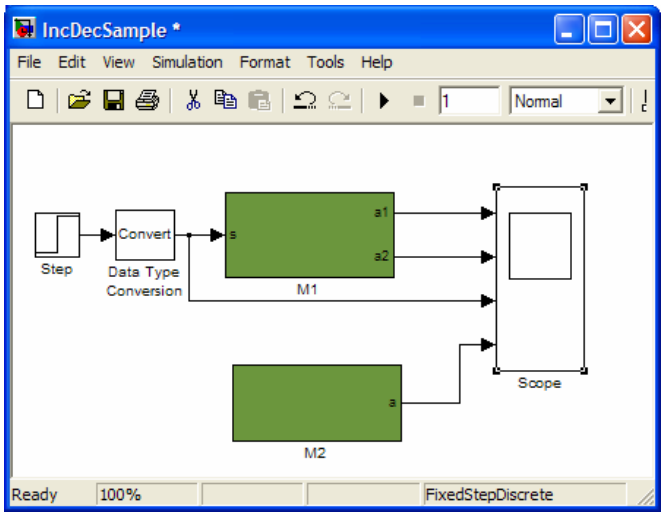
TDL source code of module *M2*

Choose the menu item File>View TDL Code menu item in the TDL:VisualCreator tool. This opens a read-only text viewer that shows the TDL program text of module *M2*:

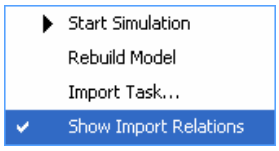


Test module *M2*

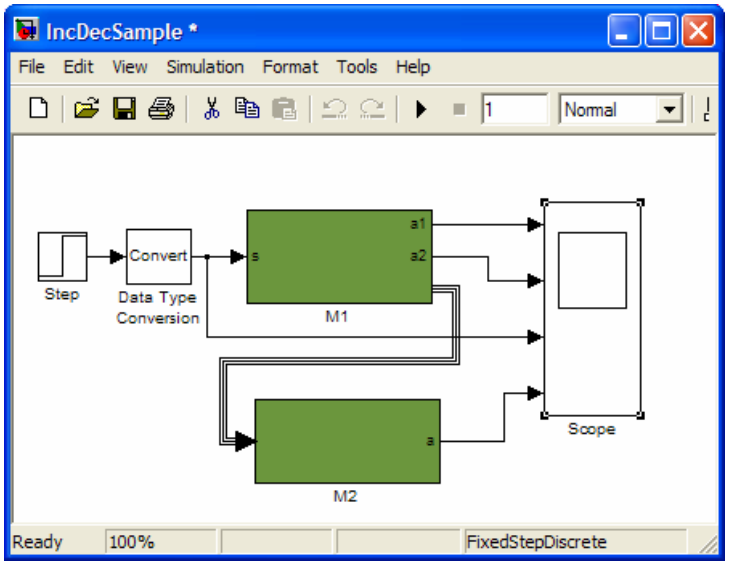
In the top-level subsystem add another input port to the Scope block to also be able to watch actuator *a* of *M2*:



In order to show the import relationship between these two modules, choose the Tools>Show Import Relations menu item in one of the TDL:VisualCreator windows.

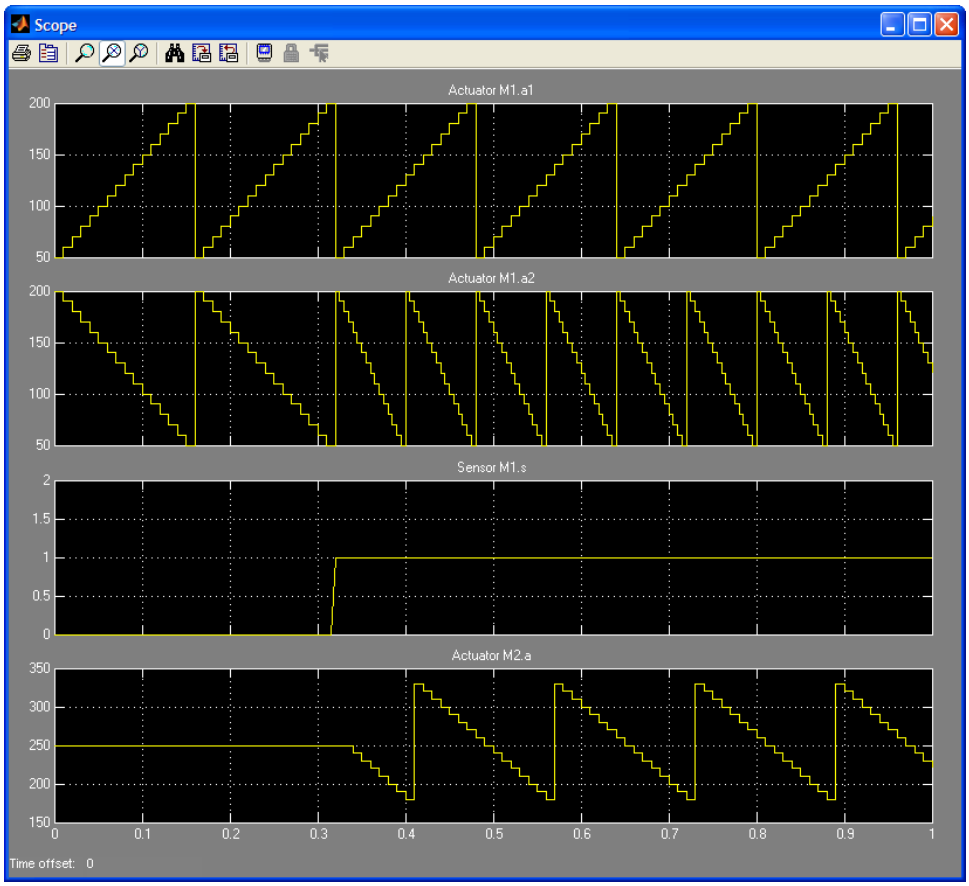


This shows the import relationship between module M1 and M2 in the form of a Simulink® bus:



Simulation. Start the simulation either by pressing the Start simulation icon in the tool bar of the Simulink® editor or choose the Start menu item in the Tools menu of the Simulink® editor (Tools>Start Simulation). This causes the TDL:VisualCreator tool to invoke the TDL:Compiler and to generate internally used Simulink® blocks that deliver the TDL semantics when the overall model is simulated.

After the simulation has finished, the Scope block should show the following output:



Synchronous actuator updates and advance calculation

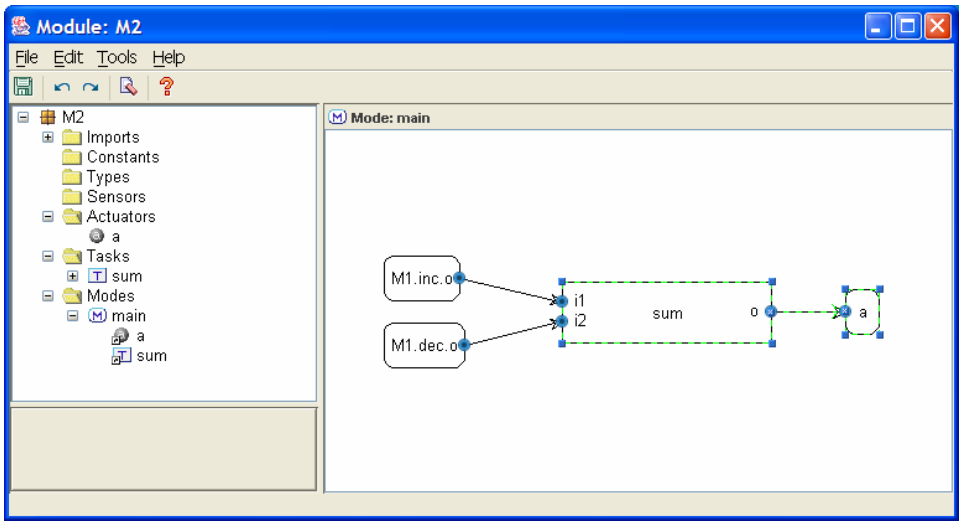
In the previously presented Scope output, the axis ‘Actuator M2.a’ shows that the timing behavior of the actuator update follows TDL’s LET semantics. This means that the actuator is set to the new value of output port ‘sum.o’ with a delay of the task invocation’s LET, which is 10 ms.

Digital controller applications, however, may require updating the actuator as soon as possible. As a rule of thumb, the task period should be at least 10 times as large as the delay between reading inputs and setting the actuator in order to get stable controller behavior. This goal can be reached on slow processors by means of so called advance calculation, which splits the calculations done by the digital controller such that a fast calculation of the actuator value can be performed before more CPU intensive operations such as computing a polynomial is carried out later. In fact the CPU intensive part for the next round is performed in the delay between setting the actuator and the next reading of the inputs.

TDL supports advance calculation and fast actuator updates by means of task sequences.

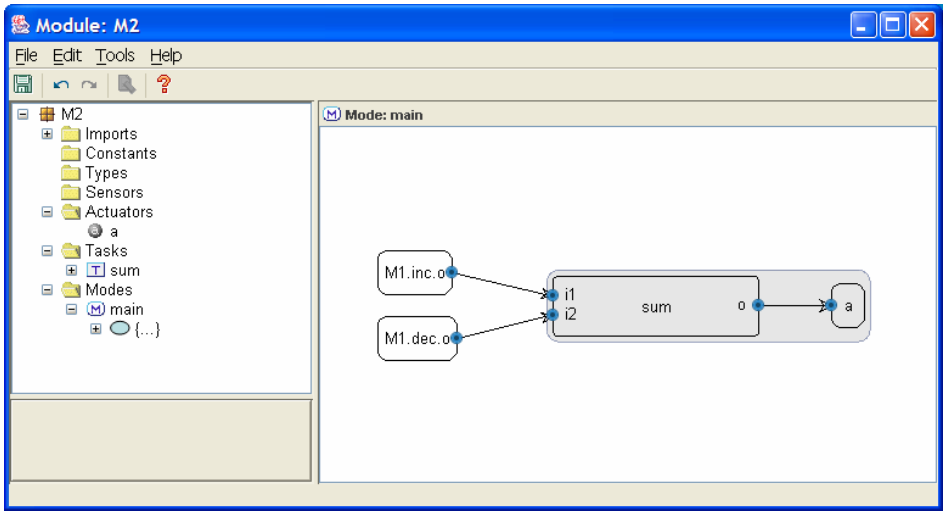
A task sequence combines a task invocation and subsequent actuator updates where the actuator updates are performed right at the release time of the invoked task, that is in logically zero time after the invocation of the task.

As an example, we apply this technique to Module M2. In order to group the invocation of task sum and the actuator update to one sequence, select both, the task sum and the actuator a in the Mode editor.

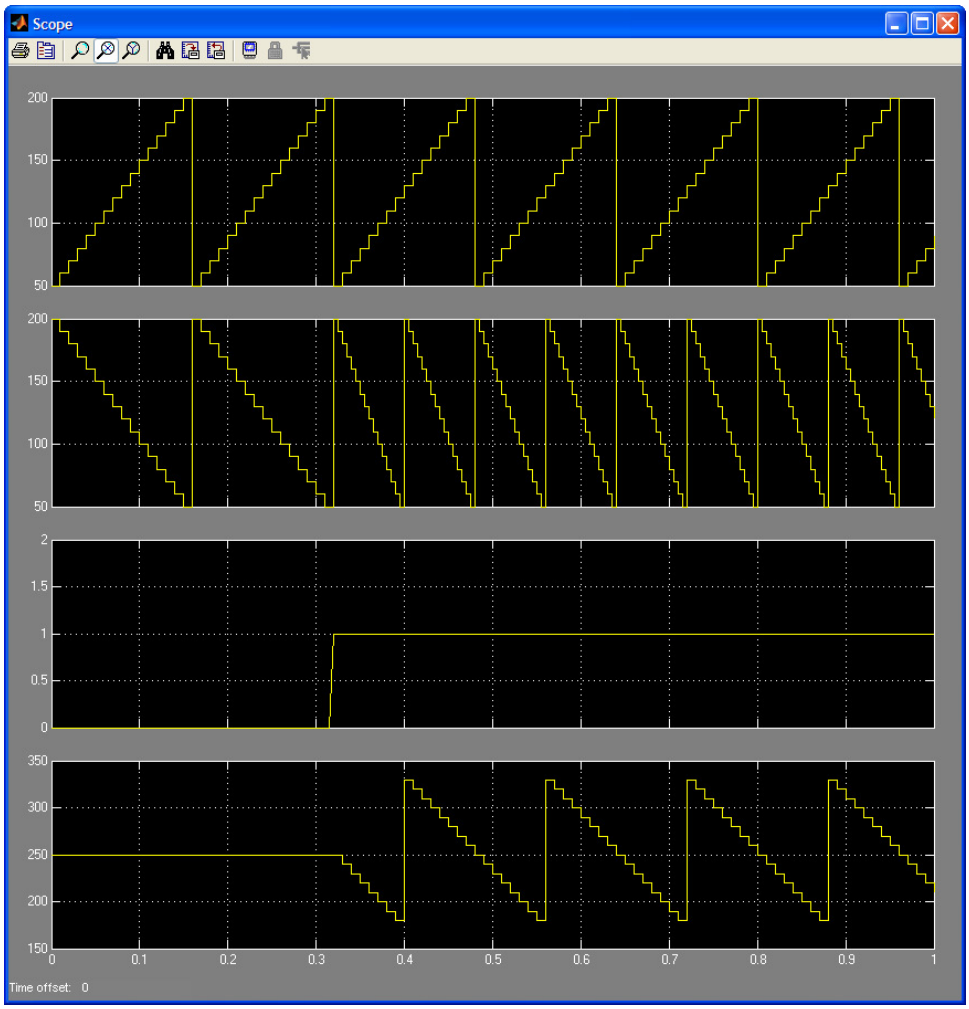


Then right-click within the Mode editor and choose the ‘Group to sequence’ item in the pop-up menu.

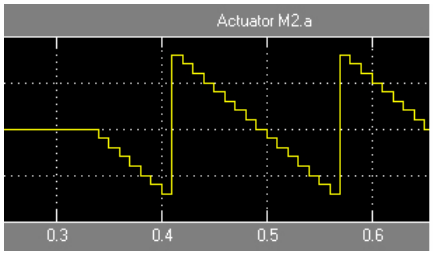
The Mode editor now groups the task sum and the actuator a in one unit and indicates the grouping with a gray bounding box:

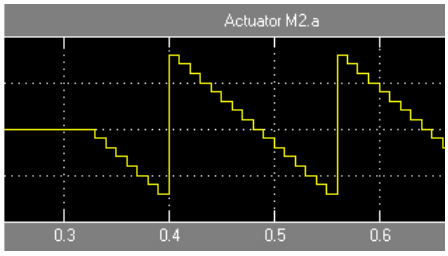


When simulating the Simulink® model again, the Scope shows the corresponding different timing behavior with an update of the actuator in module M2 that takes place 10ms earlier than in the original version:



The following screen shots illustrate the difference in detail:





Structured Data Types

In the previous example, we only used a basic type, namely integer (int). However, in practice, we often have to deal with more complex data structures. TDL supports also arrays and structures.

For the sake of clarity, we outline the usage of such data types in the context of a slightly modified variant of the example shown above. The complete Simulink® model (stored in IncDecSample2.mdl) is located in your installation directory.

IncDecSample2 is based on the previous example but uses an array type instead of the basic integer type. In this extended example, both tasks of M1, namely inc and dec, produce two values. Therefore, we define an array type.

Adding new types. New TDL types are added by right-clicking on the *Types* folder in the tree and selecting 'New Type' from the pop-up menu. This creates a new type type1. In the inspector, we can choose the 'Category' of our new type. We can either define the type to be an 'alias' for an already existing type, an 'array' or a 'struct'.

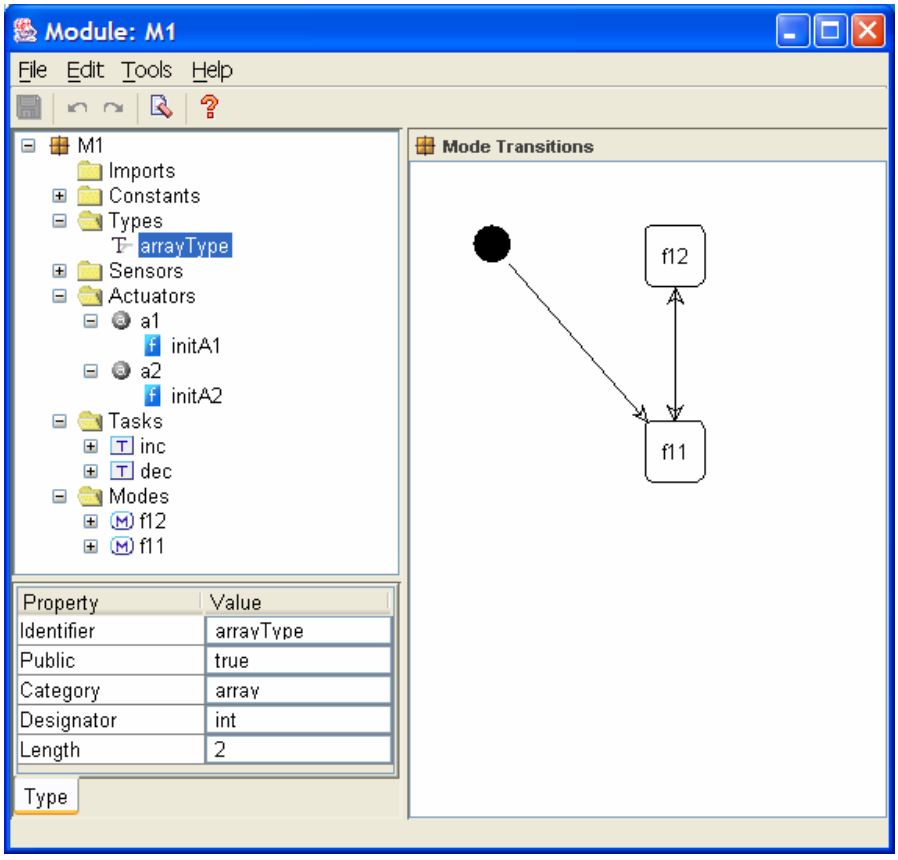
For array types, we have to specify the element's type of the array with the 'Designator' property, and its 'Length'.

In our example, we need an integer array with length 2.

Struct types, have no additional property, but would require the specification of structure members. To add a new member to a struct type, right-click on the type in the tree and select 'Add struct member' from the pop-up menu.

We use the integer array consisting of two integers for our actuators a1, a2, and for the output ports of task inc and dec in M1, as well as for actuator a and all ports of task sum in M2.

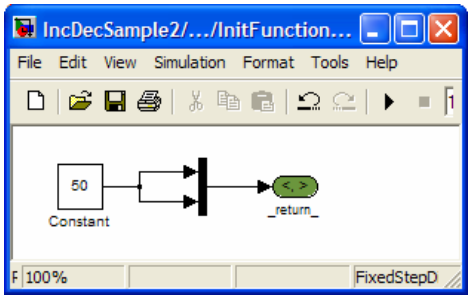
TDL ports that use an array data type expect multiplexed Simulink® signals, and ports using a struct data type expect a Simulink® bus as in- or output.



Note, that an output port or actuator of type array or struct cannot be initialized by a value defined with its 'Initial Value' property. An initializer function must be used instead.

Creating initializer functions. To create an initializer function for a port, right-click on the port in the tree and select 'New init function' from the pop-up menu.

Analogous to task- or guard-functions, initializer functions are specified in Simulink®:



We have to create initializer functions for all actuators and output ports in our example.

This short summary of structured types concludes the guided tour of the TDL:VisualCreator tool. We recommend that you experiment with the tool yourself. For example, you could extend the original inc-dec case study and replace the basic int type by an array as sketched above and comapre your solution to the one provided by us as IncDecSample2.mdl. Of course, we encourage you also to transform existing models to TDL. A case study, called Active Rear Steering, was modeled based on existing controller functionality.

Generating platform-independent code for TDL modules

Once a module provides the required behavior, the TDL:VisualCreator supports the export of the TDL source code and the generation of C code for each task function of a TDL module, currently with MathWorks' Real-Time Workshop Embedded Coder tool.

In order to export the TDL source code select the File>Export TDL Code menu item and specify the directory in which the TDL source code file named according to the module name should be written.

In order to export the C code for each task function of a TDL module select the File>Export Functionality Code menu item and specify the directory in which the C source code files named according to the tasks should be written.

Platform-specific code is generated with the TDL:VisualDistributor tool. For the detailed description how to work with the TDL:VisualDistributor we refer to its documentation.

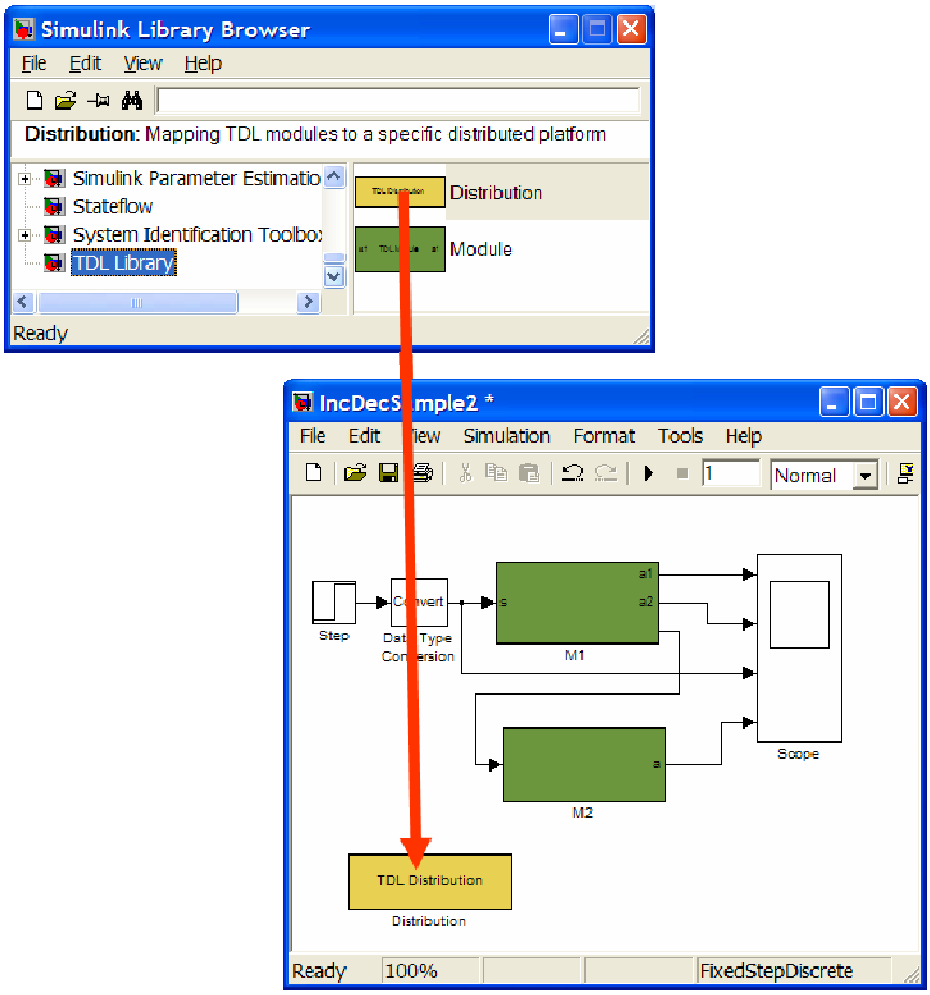
Transparent distribution of TDL modules

Remember that TDL modules can be executed on a single node or can be distributed to several nodes in a distributed system without changing the observable behavior of the TDL modules and especially without considering the distribution of the modules in the TDL modeling phase. We assume that we want to distribute the TDL modules *M1* and *M2* on a FlexRay cluster with two nodes *node1* and *node2*.

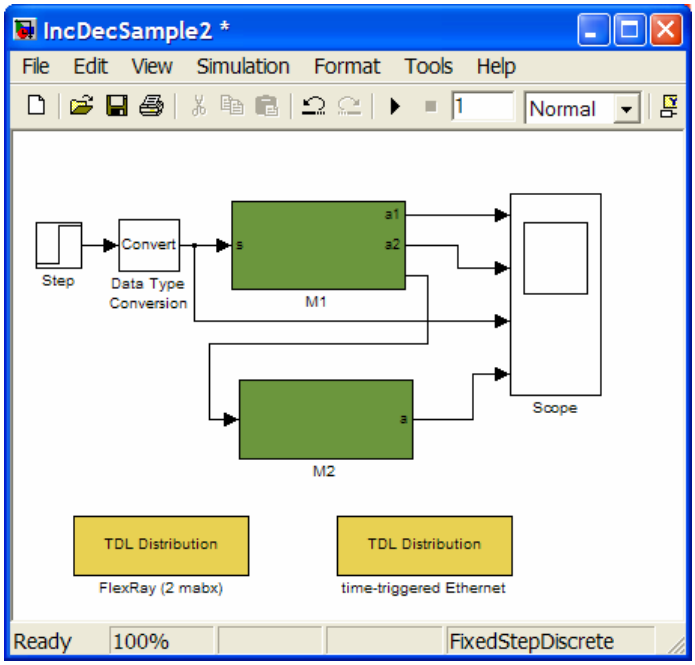
The TDL:VisualDistributor allows (1) the definition of the bus topology (2) the mapping of TDL modules to the particular nodes that are connected via the bus and (3) the platform-specific code generation. For a detailed documentation of how to

use the TDL:VisualDistributor we refer to its description which comes with the TDL:VisualDistributor.

For each Simulink model that contains one or more TDL modules any number of platform mappings can be defined. For defining a platform mapping, drag the Distribution block from the Simulink library browser to the model:

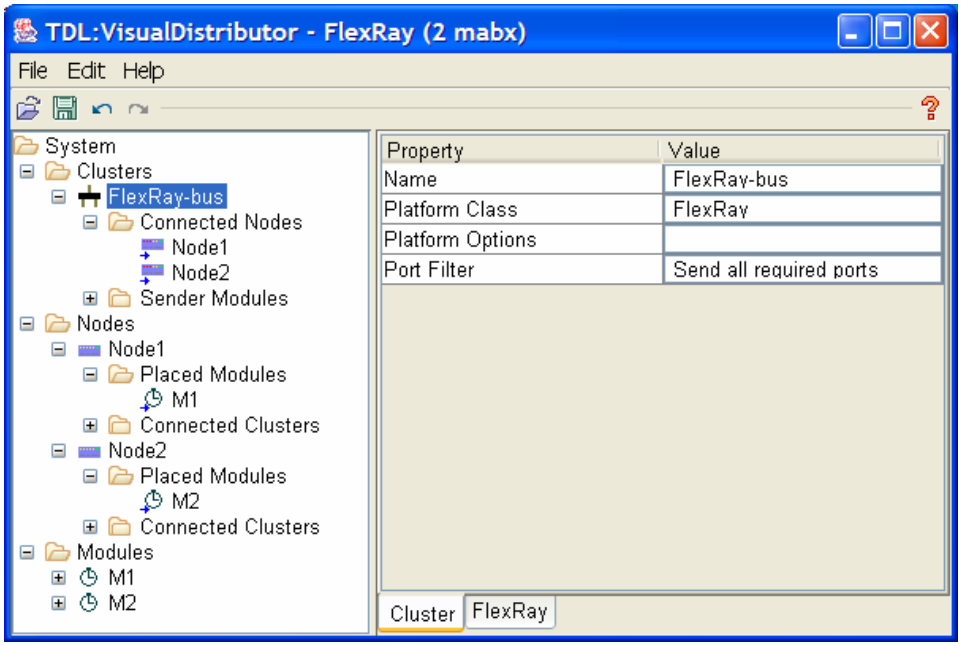


Note that you can define any number of mappings by dragging more Distribution to your model:



The following screen shot exemplifies one TDL module-to-node mapping by means of one TDL:VisualDistributor instance for FlexRay. The TDL modules specified in the model are listed in the Modules folder.

We assume that the platform and topology have already been defined. Then the developer simply drags a module listed in the Modules folder to the node where it should be executed. In the example, *Node1* executes module *M1*, and *Node2* executes module *M2*. The two nodes are connected to a bus called *FlexRay-bus*.



All necessary code to execute the TDL modules M1 and M2 is then generated for the particular platform, if the platform offers sufficient communication and computing resources. If this is the case, the TDL tools guarantee that the observable behavior of the TDL components on the particular platform is identical to the behavior in the

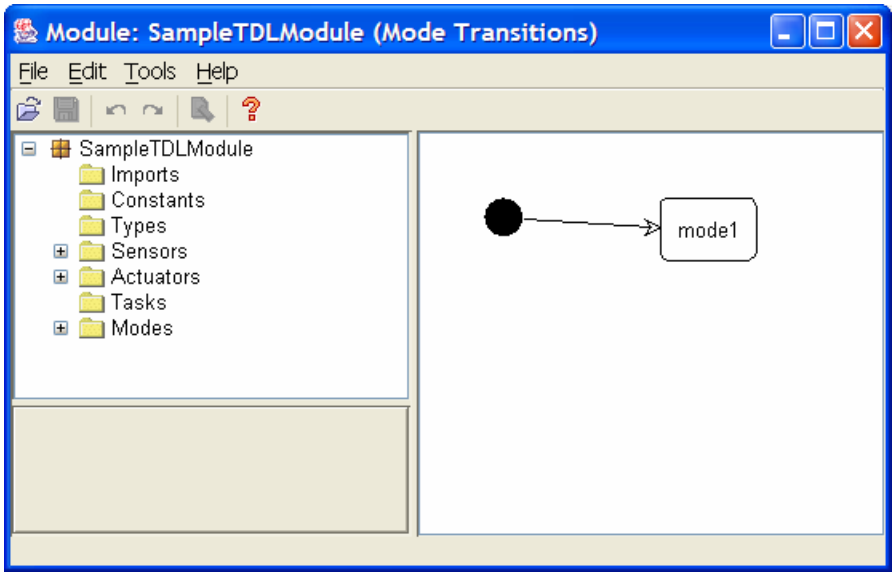
simulation. If a platform offers not sufficient communication and computing resources for the specified timing behavior no code is generated.

TDL:VisualCreator

This part of the documentation presents the features of the TDL:VisualCreator, a syntax-driven visual and interactive editor for TDL programs. If you went through the guided tour you already know most of it.

The TDL:VisualCreator tool is an alternative to entering TDL programs as text. It can be used as stand-alone tool instead of a text editor. Once you have modeled your TDL program you can generate the TDL source text. You can also view the textual version of a TDL program at any time. The TDL:VisualCreator tool can also be used within the Simulink® environment. In this case you can model the functionality, in particular the TDL task function implementations and guard functions, with standard Simulink® blocks and you can simulate the overall system.

No matter whether you use the TDL:VisualCreator as stand-alone tool or integrated into Simulink®, the user interface is the same as shown in the following screen shot after starting the TDL:VisualCreator tool:



We will first explain how you set up the tool as stand-alone tool or as an add-on to Simulink®. Then we present the preliminaries to understand the overall editor concept and continue to present how each TDL construct is specified with the TDL:VisualCreator tool.

Differences between the demonstration and product versions

The demonstration version of the TDL:VisualCreator tool has no restrictions regarding editing and saving of your models, though the demonstration version might not support all the features of the current product version. However, the demonstration version canot be combined with the TDL:VisualDistributor tool so that you cannot generate executables for specific platforms. What you can do is

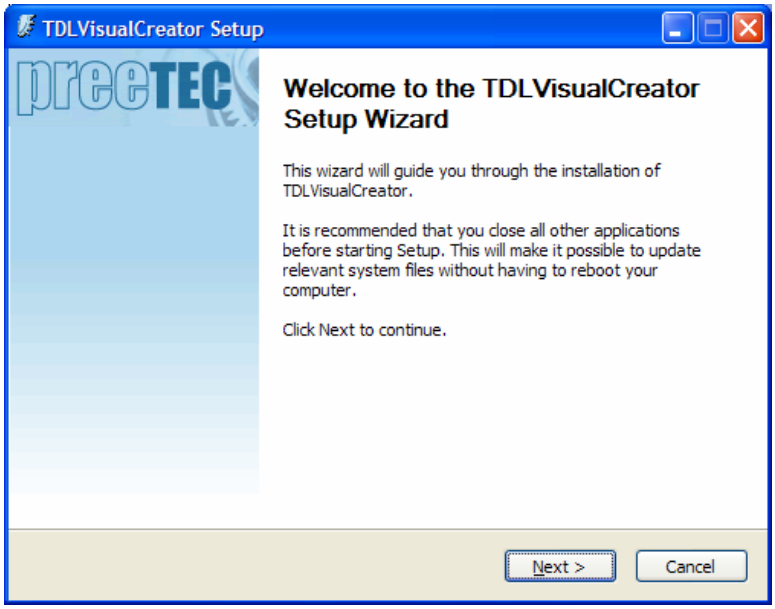
simulate the behavior in Matlab/Simulink. This should be sufficient for experimenting with TDL as we guarantee that the behavior in the simulation is equivalent with the behavior on supported execution platforms, provided that the platform offers sufficient computing and communication resources for the timing behavior specified in the TDL modules.

Technically, the main difference between the demonstration version of the TDL:VisualCreator and its product version is that the demonstration version stores the models in a specific format (let us call it demo format). Files in the demo format can be read by the product version of the TDL:VisualCreator tool. So you can easily migrate your models to the product version, but you cannot transfer models from the product version back to the demo format.

For using the product version you need a hardware-lock (dongle). The demonstration version executes without a hardware-lock (dongle).

Set-up as stand-alone tool

Execute the TDLVisualCreatorSetup.exe file. The installation wizard will guide you through the installation.


TDLVisualCreator Setup
preeTEC
Welcome to the TDLVisualCreator Setup Wizard
This wizard will guide you through the installation of TDLVisualCreator.
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.
Click Next to continue.
Next >
Cancel

For starting the TDL:VisualCreator tool, connect the hardware-lock (dongle) to a free USB port and select the TDLVisualCreator entry from your start menu. The demonstration version does not require a hardware-lock.

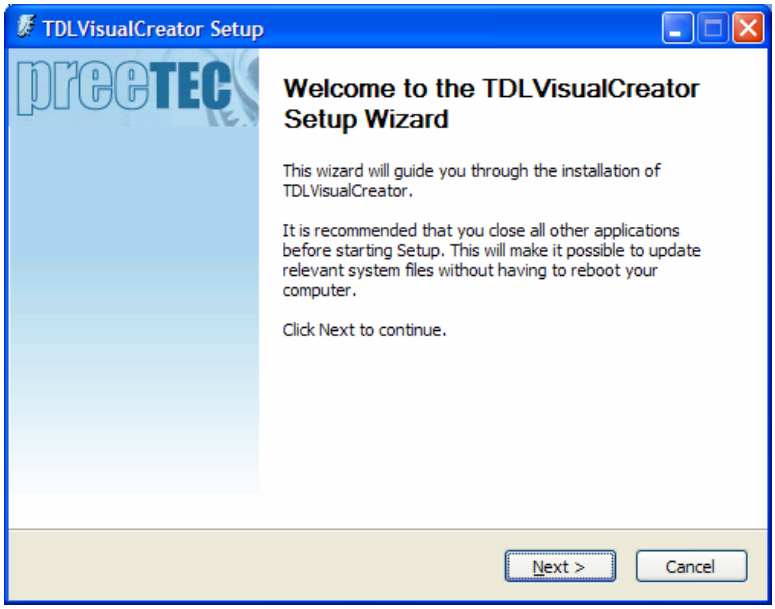
For working with the TDL:VisualCreator (product and demonstration versions) you need the Java 2 Platform, Standard Edition, v 1.4.2 (J2SE, download it from http://java.sun.com/j2se/1.4.2/download.html), for executing the TDL:VisualCreator tool.

Set-up as add-on to Simulink®

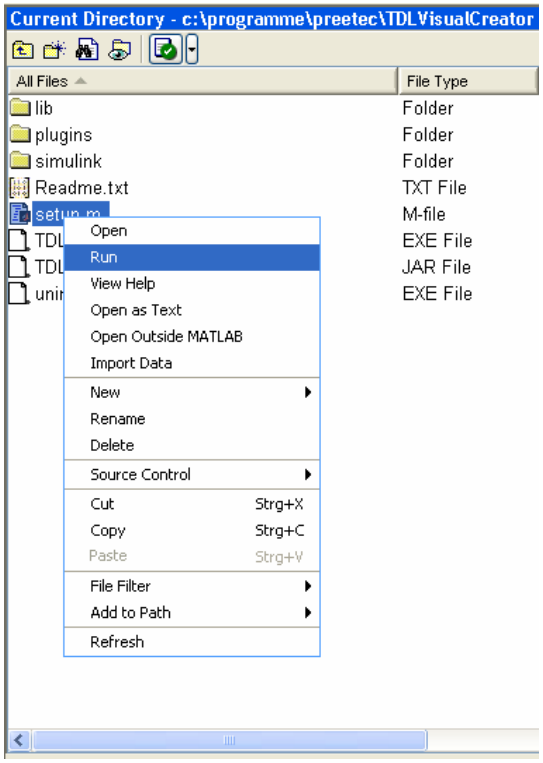
We have tested the TDL:VisualCreator tool with Matlab® version 7.0 (R14) and 7.1 (R14SP3), Simulink® version 6.0 (R14) and 6.3 (R14SP3), Real-Time Workshop® version 6.0 (R14) and 6.3 (R14SP3), and Real-Time Workshop Embedded Coder® version 4.0 (R14) and 4.3 (R14SP3).

Thus, we recommend to use the TDL:VisualCreator tool within these environments.

Execute the TDLVisualCreatorSetup.exe file. The installation wizard will guide you through the installation.

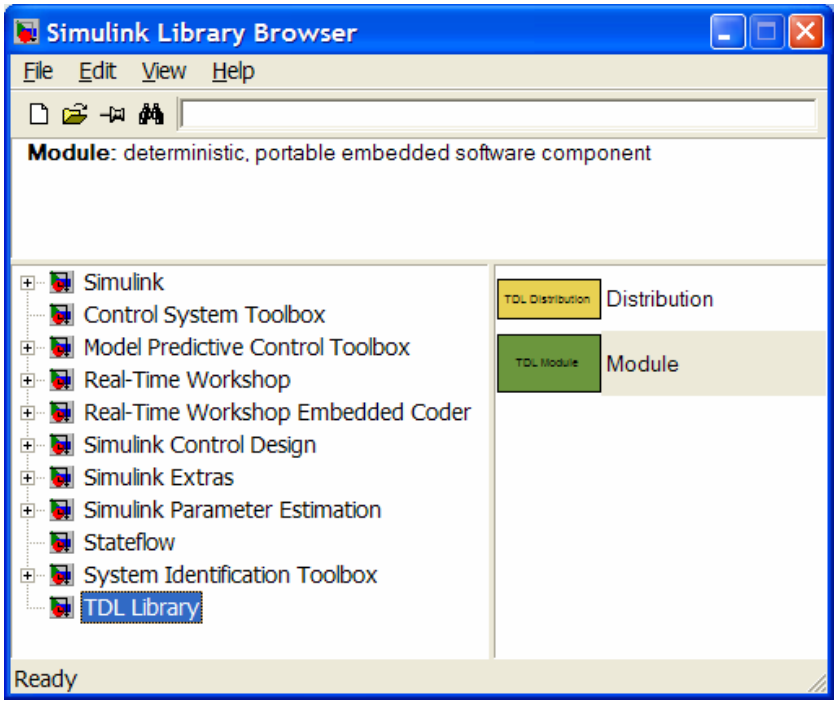

TDLVisualCreator Setup
preeTEC
Welcome to the TDLVisualCreator Setup Wizard
This wizard will guide you through the installation of TDLVisualCreator.
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.
Click Next to continue.
Next >
Cancel

Then start Matlab®, change to the TDL:VisualCreator installation directory and start the setup.m script.



Connect the hardware-lock (dongle) to a free USB port (in case of using the product version; for the demonstration version you do not need a hardware-lock), restart Matlab® and open the Simulink® Library Browser. Then the TDL Library should be listed as shown in the following screen shot. When you select it you see the TDL

Module block (green box) and the TDL Distribution block (yellow box). In case of the demonstration version no Distribution block is listed in the TDL Library.



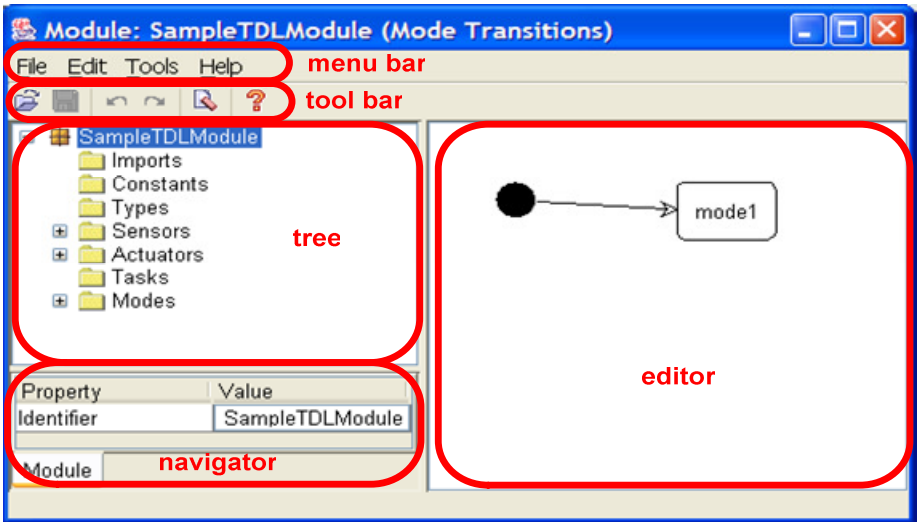
After adding the Module block to a Matlab®/Simulink® model you double-click on that block and the corresponding TDL:VisualCreator opens for editing the particular TDL module.

If you need to map the TDL module(s) to a specific platform you drag one Distribution block for each mapping into the Matlab®/Simulink® model. With a double-click you open the TDL:VisualDistributor tool, with which you can specify the platform, map the TDL modules to that platform and generate platform-specific code. For details we refer to the documentation of the TDL:VisualDistributor tool.

Preliminaries

The TDL:VisualCreator is a syntax-driven tool for visually editing syntactically correct TDL programs. One instance of the TDL:VisualCreator allows you to model one TDL module.

The following figure shows the principal elements of the TDL:VisualCreator tool window: The tool bar is under the menu bar. The window is split in three panes, the tree pane and the navigator pane on the left side as well as the editor pane on the right side. For short we call the tree pane just tree, the navigator pane simply navigator, and the editor pane the editor.



Tree and navigator. The tree (left pane of the TDL:VisualCreator window) shows the module subtree with the module name as the root item and all possible TDL constructs as subnodes: import declarations, constants, types, sensors, actuators, tasks and modes.

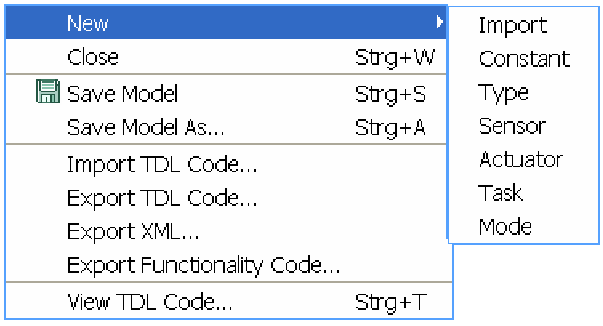
If you add an instance of one of these constructs to the module the corresponding subnode lists that instance. The properties of most TDL constructs are specified via the navigator.

Editor. The editor (right pane of the TDL:VisualCreator tool window) provides the editing capabilities for certain aspects of TDL modes: the TDL developer models socalled mode transitions, task invocations, actuator updates and port assignments in the editor. We explain the editing of TDL modes in a separate section below.

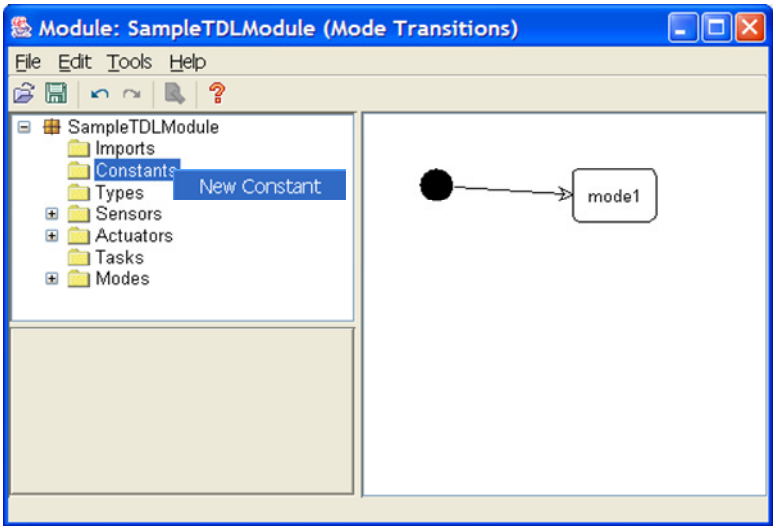
Adding a TDL construct

No matter which TDL construct you want to add to a TDL module you can accomplish that in one of the following alternative ways:

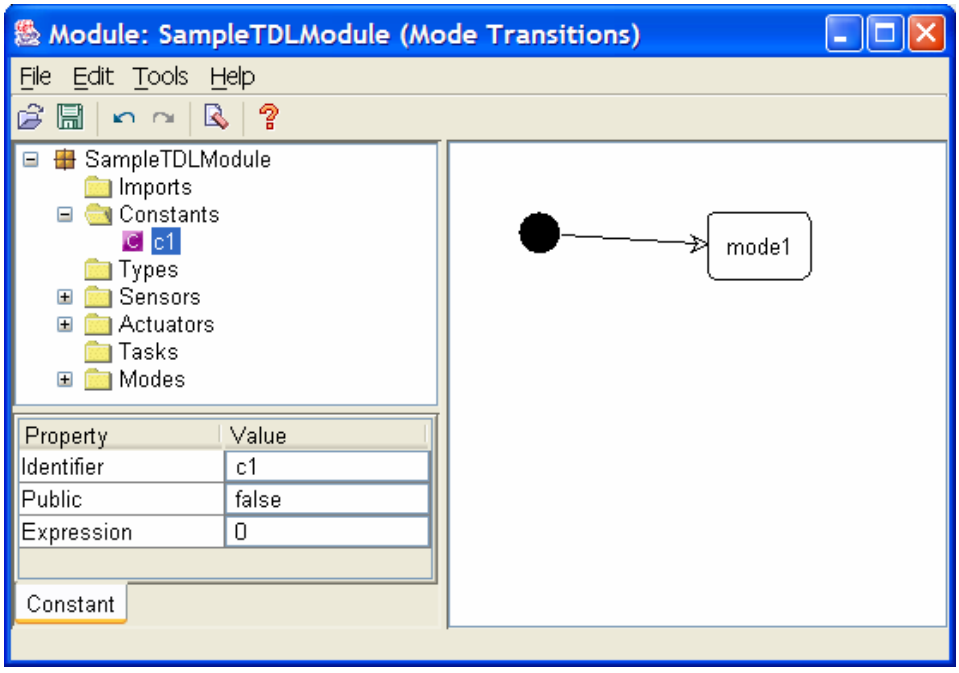
- Choose the File>New>*TDL_construct* menu item where *TDL_construct* is a place holder for any TDL construct: import declaration, constant declaration, type declaration, sensor declaration, actuator declaration, task declaration, mode declaration.



- Right-click on the folder corresponding to the TDL construct you want to add. This opens a pop-up menu with the menu item New *TDL_construct*. Choose the menu item.



After that the newly added TDL construct is selected and its properties are shown in the navigator, in which you can edit them. For example, the following screen shot illustrates the properties of a newly added TDL constant declaration:

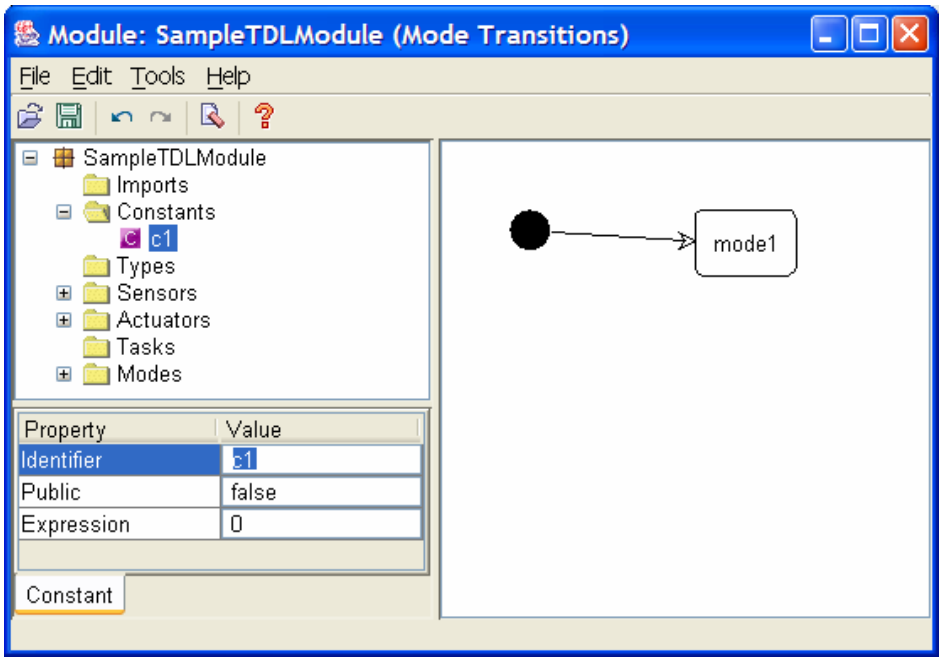


Editing a TDL construct

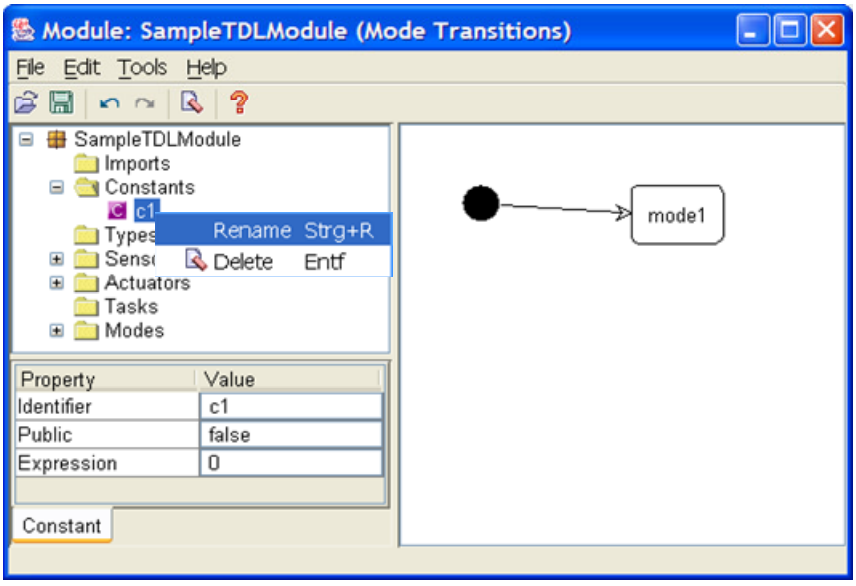
All instances of TDL constructs are listed by their name (Identifier) in the corresponding folders in the tree. To edit the properties of a particular instance of a TDL construct open the corresponding folder by clicking on the + symbol left to that folder or double-click on the folder.

Then select the particular item by single-clicking on it in the tree to see its properties in the navigator. Editing modes and editing tasks is more complex and explained in the particular sections below.

If you only want to change the name (Identifier) you can select the item with a left-click and edit its name in the navigator. The following sample screen shot illustrates this for constant c1:

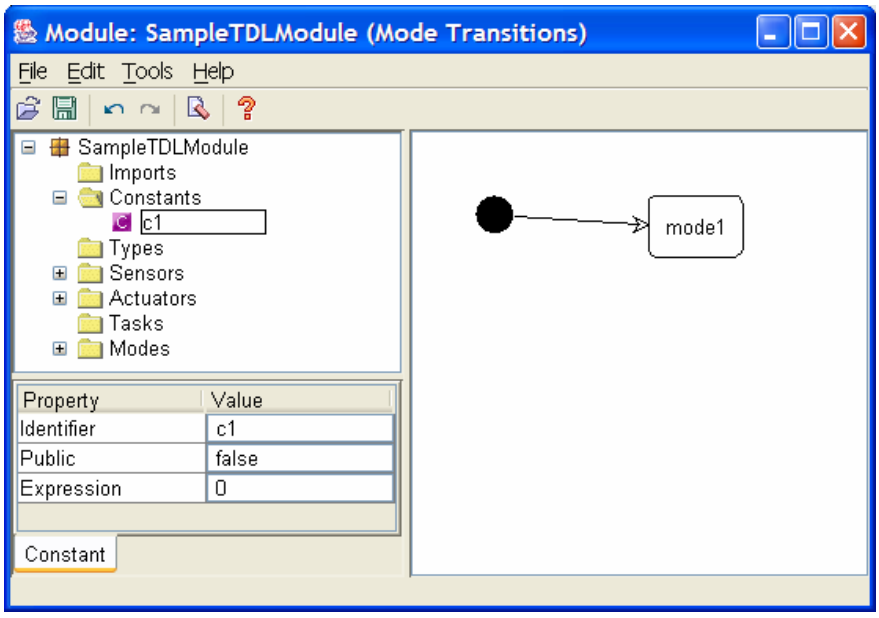


Alternatively you can right-click on the item in the tree and choose the Rename pop-up menu item. The following sample screen shot illustrates this for constant c1:

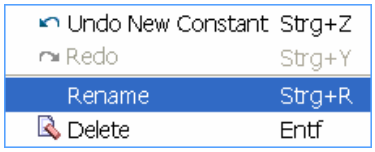


The

n you can edit the name directly in the tree:

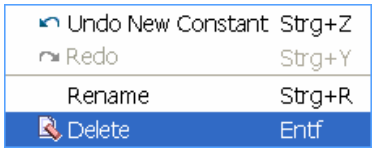


Alternatively you can also select the particular item in the tree and choose the Edit>Rename menu item:

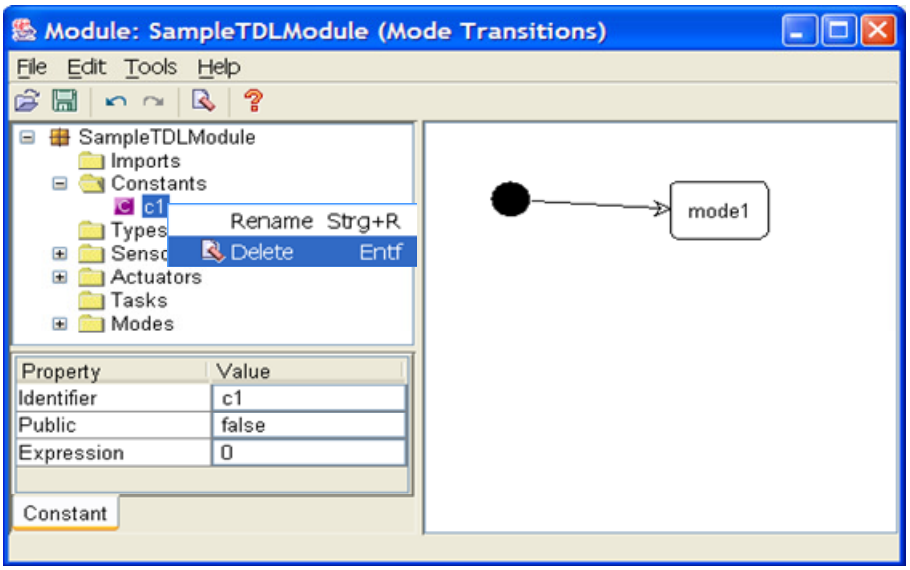


Deleting a TDL construct

Select the particular item in the tree and choose the Edit>Delete menu item.



Alternatively you can right-click on the item in the tree and choose Delete from the pop-up menu.



TDL program text

At any time you can generate the textual representation of the TDL module you are editing in the TDL:VisualCreator: Choose the File>View TDL Code... menu item for that purpose. A read-only editor view displays the TDL program text.

The TDL:VisualCreator tool also supports the importing of a TDL program text (File>Import TDL Code...) and the exporting of a TDL program text (File>Export TDL Code...).

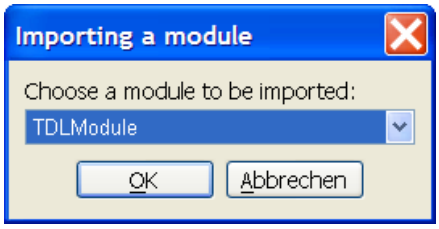
TDL constructs

The following sections explain the properties of the TDL constructs import declaration, constant declaration, type declaration, sensor declaration, actuator declaration, and task declaration. Further sections explain how to model tasks and modes.

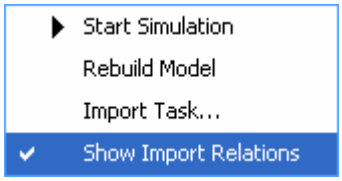
TDL Import declaration

In case of using the TDL:VisualCreator as stand-alone tool, another module can only be imported if it was saved with the TDL:VisualCreator in the TDL:VisualCreator format. A file dialog allows the selection of the particular module.

In case of using the TDL:VisualCreator tool within Simulink®, the TDL modules that are available for an import need to be placed in the same Simulink® model as the module that wants to import them. The following dialog box exemplifies how an import declaration is specified if the TDL:VisualCreator tool is used within Simulink®:



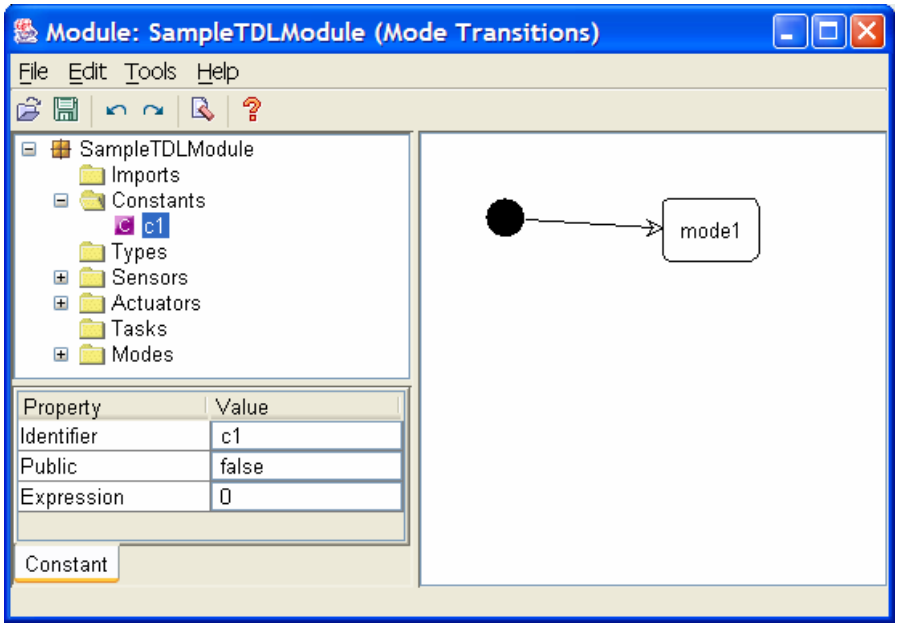
In order to visualize existing import relations between TDL modules within a Simulink® model, choose the Tools>Show Import Relations menu item:



For further details we refer to the EBNF definition of this TDL construct.

TDL Constant declaration

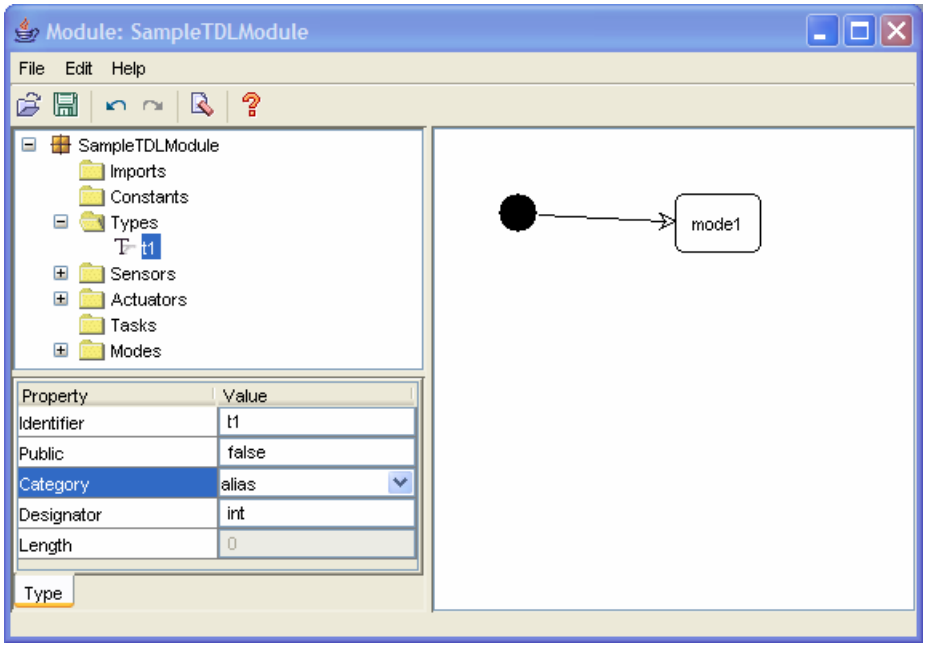
A constant declaration requires the definition of the following properties: Identifier (name), Public (enable or disable access from other modules), Expression (value of the constant). The properties are listed in the navigator:



For further details we refer to the EBNF definition of this TDL construct.

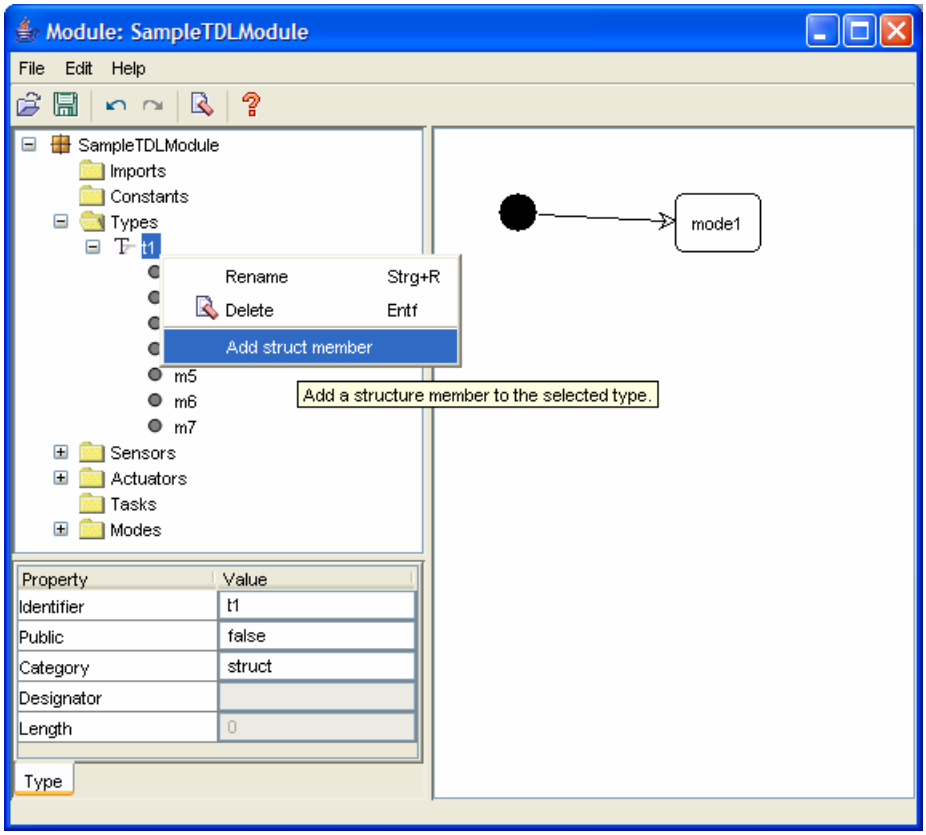
TDL Type declaration

A type declaration requires the definition of the following properties: Identifier (name), Public (enable or disable access from other modules) and Category. Depending on the Category value, additional properties may have to be defined:



Categories:

- alias
  Alias types require the definition of the supplemental property Designator (basic- or user-defined type).
- array
  Array types additionally require also the Designator and Length property to be set.
- struct
  Struct types need no additional properties. Elements are added by right-clicking on the type in the tree and require the definition of the Identifier (name) and Type property.

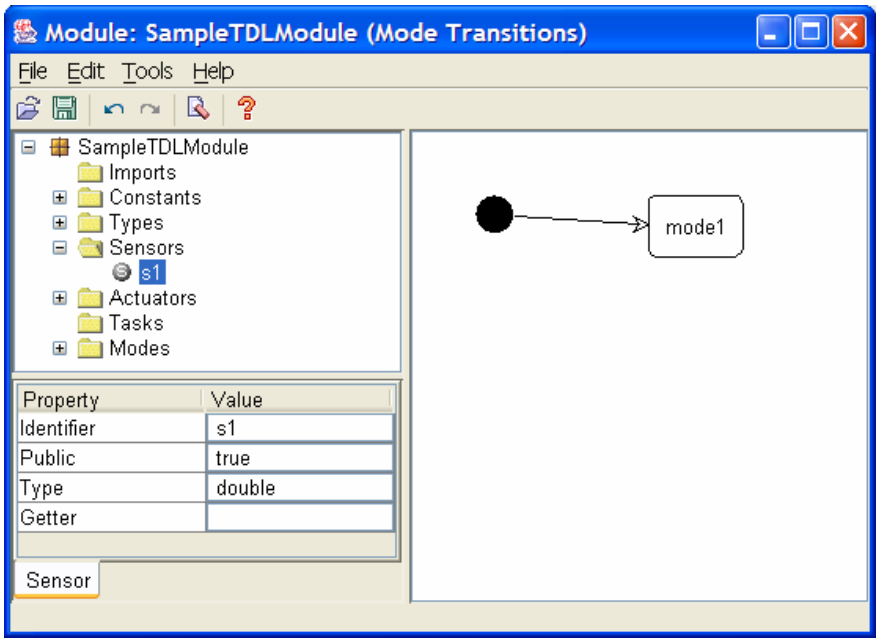


In case of using the TDL:VisualCreator within Simulink®, a signal connected to an array-typed port is expected to be multiplexed (combined signals of the same type). A struct-typed port expects a Simulink® bus as in-/output.

TDL provides a set of basic types, which matches those found in the programming language Java. For details check the description in the language report.

TDL Sensor declaration

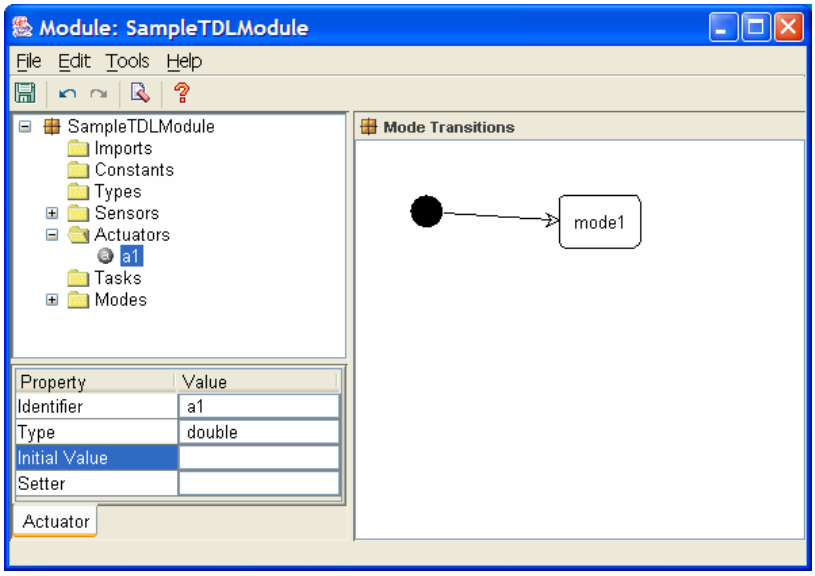
A sensor declaration requires the definition of the following properties: Identifier (name), Public (enable or disable access from other modules), Type (choose one of the supported types from a pop-up menu in the Value column), Getter (name of an external getter function). If you do not specify a getter function the default name get*SensorName* is used, where SensorName is the Identifier starting with a capital letter. In the following example this would be getS1, as the Identifier has the value s1. The properties are listed in the navigator:



For further details we refer to the EBNF definition of this TDL construct.

TDL Actuator declaration

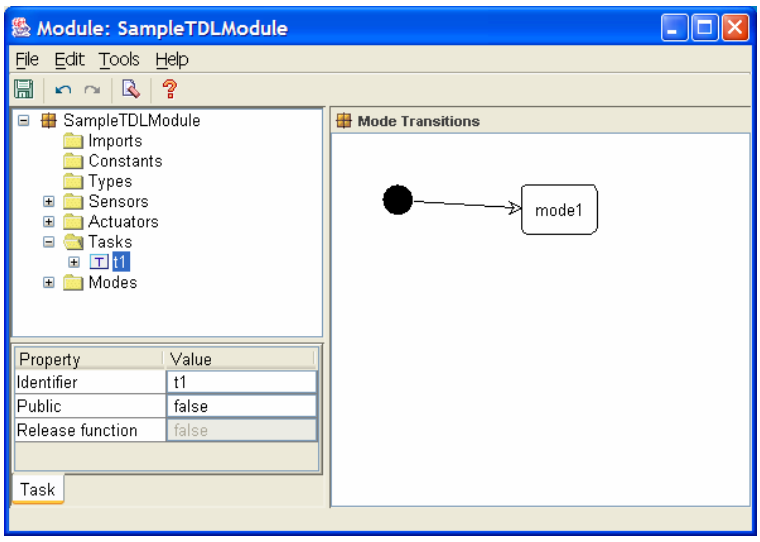
An actuator declaration requires the definition of the following properties: Identifier (name), Type (choose one of the supported types from a pop-up menu in the Value column), Setter (name of an external setter function), Initial Value. If you do not specify a setter function the default name set*SensorName* is used, where *SensorName* is the Identifier starting with a capital letter. In the following example this would be setA1, as the Identifier has the value a1. The properties are listed in the navigator:



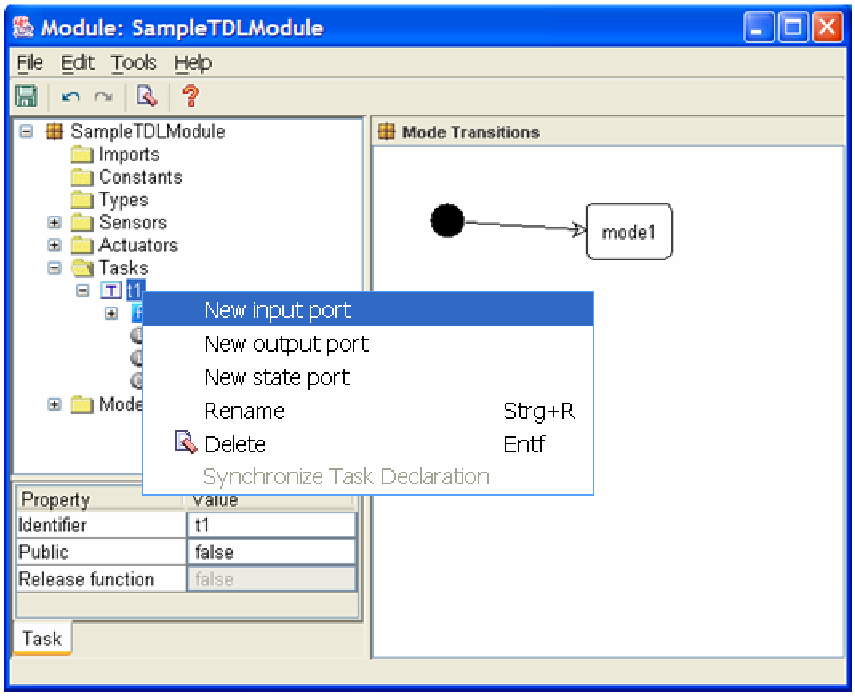
For further details we refer to the EBNF definition of this TDL construct.

TDL Task declaration

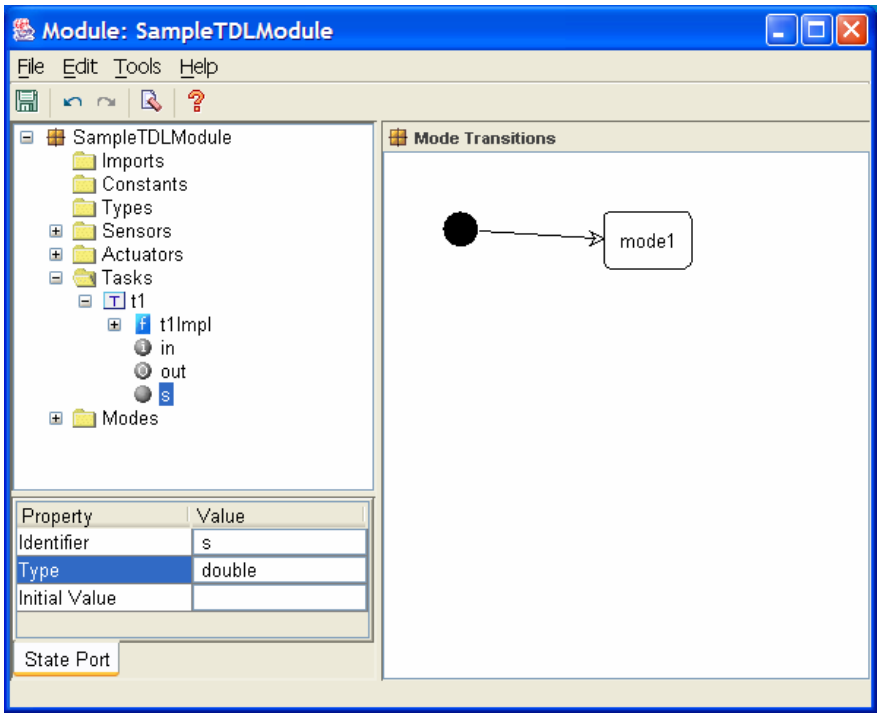
We first present the task declaration when we use the TDL:VisualCreator tool within Simulink®: In this case a task declaration requires the definition of the following properties: Identifier (name), and Public (enable or disable access to task output ports from other modules). The property 'Release function' is automatically set, depending on whether the task is part of a task sequence or not. The properties are listed in the navigator:



Task ports (input, output or state ports) are added by right-clicking on the task in the tree:

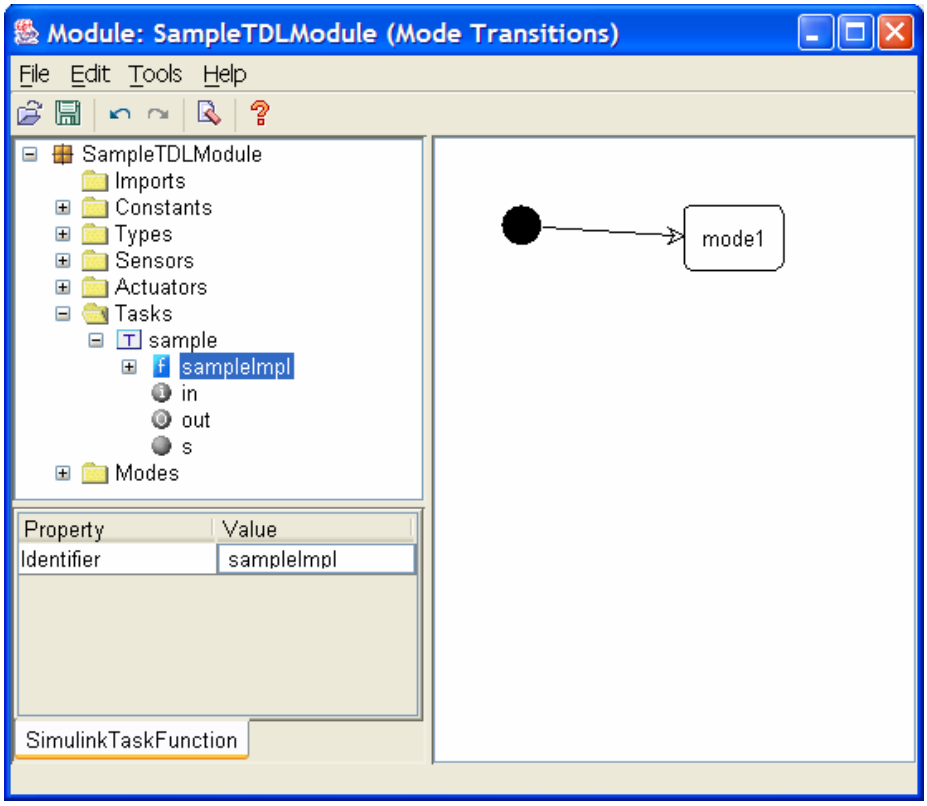


The properties of a port (Identifier and Type for input ports; Identifier, Type, Initial Value and Initializer for output and state ports) are shown in the navigator when a port is selected:



To remove a port select the task port in the tree, right-click for the pop-up menu and select the Delete menu item.

In case of using the TDL:VisualCreator within Simulink®, the task function can be modeled with Simulink® blocks by double-clicking on the <*task*>*Impl* item in the Tasks folder of the tree, where <task> is a place holder for the task name. The following screen shot shows a task *sample* with the *sampleImpl* item:



For further details we refer to the EBNF definition of this TDL construct.

TDL Mode declaration

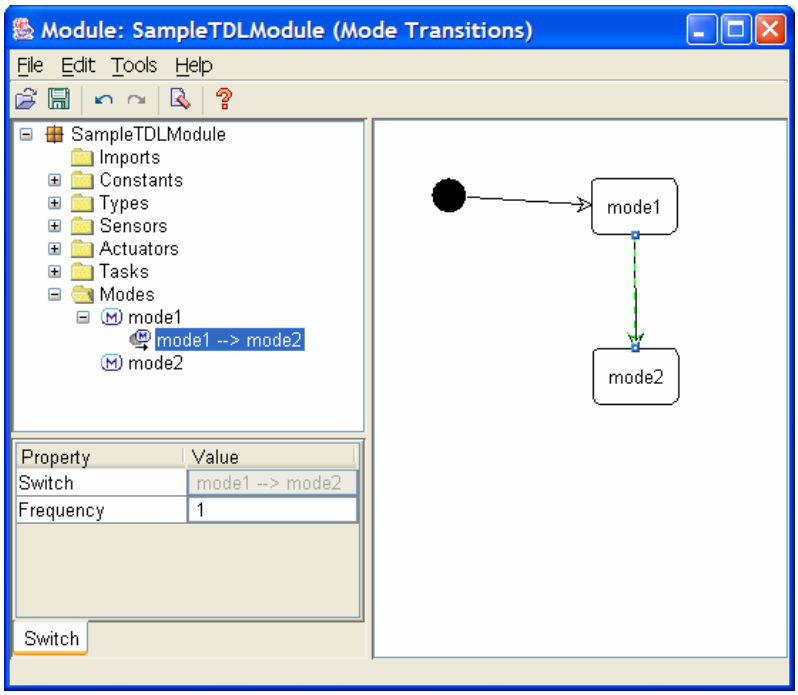
The editing of TDL modes requires up to three steps. The TDL:VisualCreator provides a different editor for each step in the editor:

- The **Mode Transition editor** is the default editor. It shows which mode is the **start mode**, indicated by the black bullet hole and the arrow from that bullet hole to the start mode. You can let the arrow point to any mode to define that the particular mode is the start mode. In order to accomplish that you draw a new arrow from the bullet hole to a mode represented by a box.

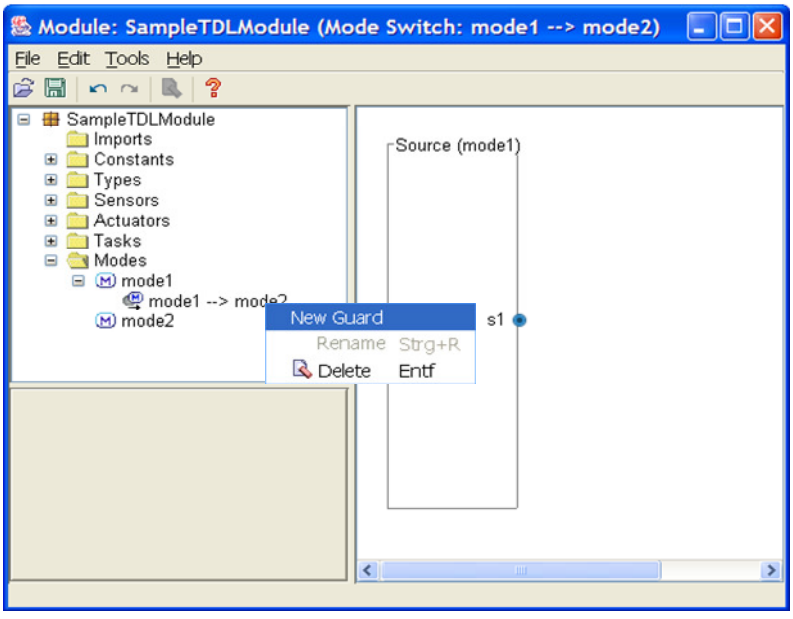
  To be able to draw a connecting arrow the mouse pointer has to be a hand when you are over the bullet hole or a mode box. If the mouse cursor is a cross, you can move a box or the bullet hole. The mouse pointer is a cross if you are in the center area of a box or the bullet hole. Otherwise the mouse pointer is a hand.

  Furthermore, arrows between modes show which mode switches are possible. For each such selected **mode switch** arrow the navigator shows the name and you can edit the Frequency property with which

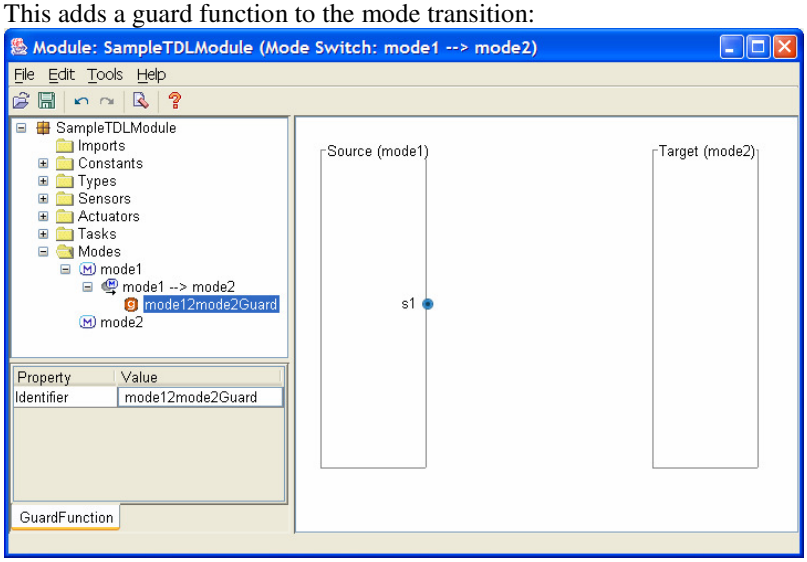
the condition is checked in the source mode.



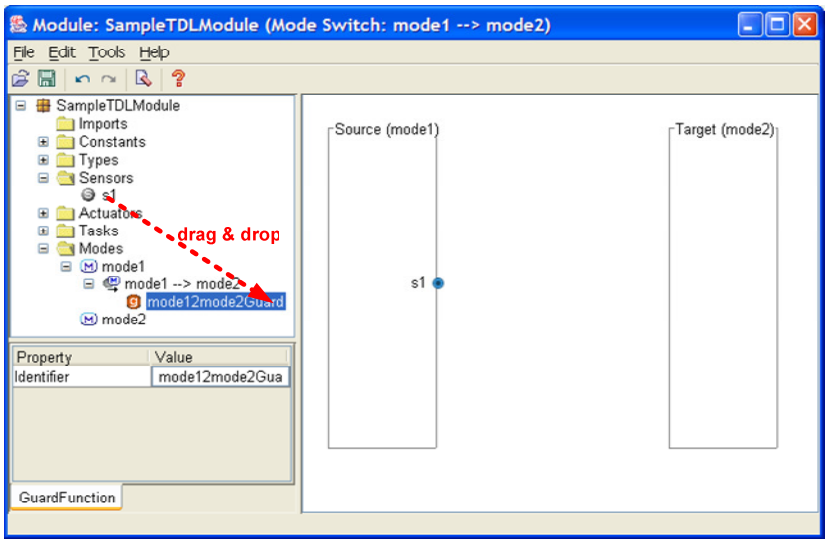
- You edit the condition by right-clicking on the mode transition item in the tree (labeled mode1 --> mode2 in the example) and selecting the menu item New Guard in the pop-up menu:



This adds a guard function to the mode transition:

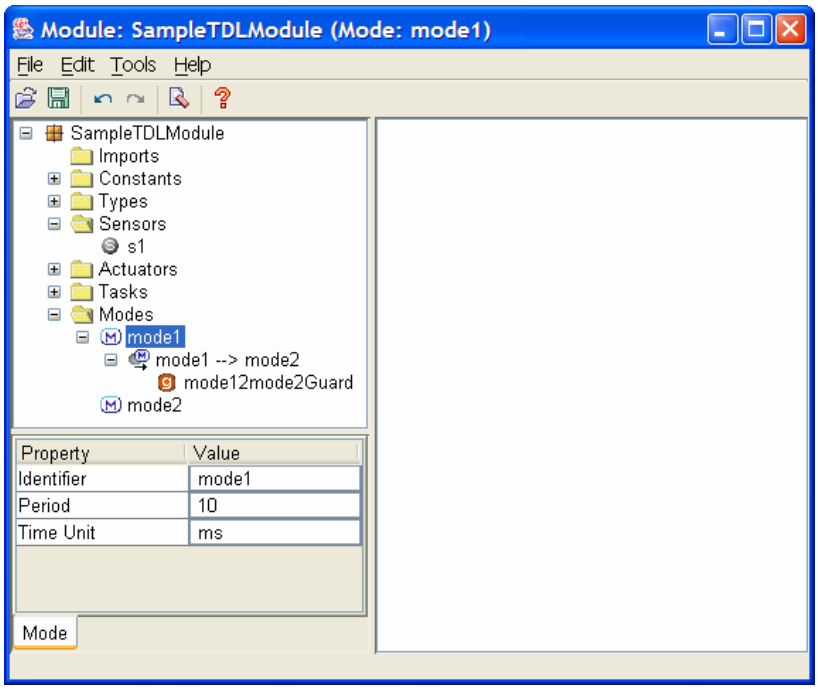


- A double-click on the guard item in the tree opens a Simulink® editor that allows you to specify the condition when a mode switch occurs. If you need a sensor value or a task output port for that condition specification, simply drag the particular sensor from the Sensors folder in the tree or the task output port from the particular task folder in the tree to the guard function. The following screen shot exemplifies this for a sensor:

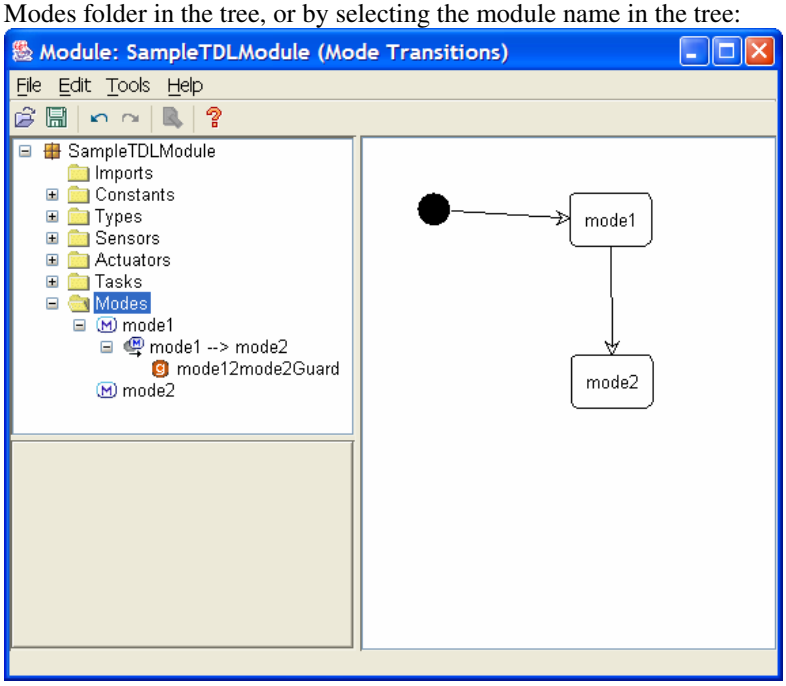


- You define the **mode period** by selecting a mode in the Modes folder in the tree and setting the Period property in the navigator. The following example shows that the mode period of *mode1* is set to 10

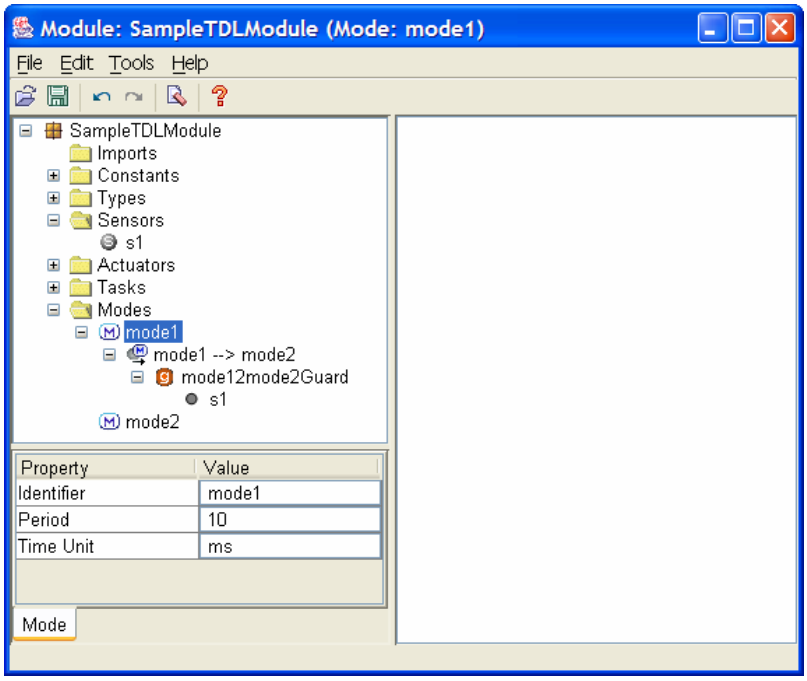
ms:



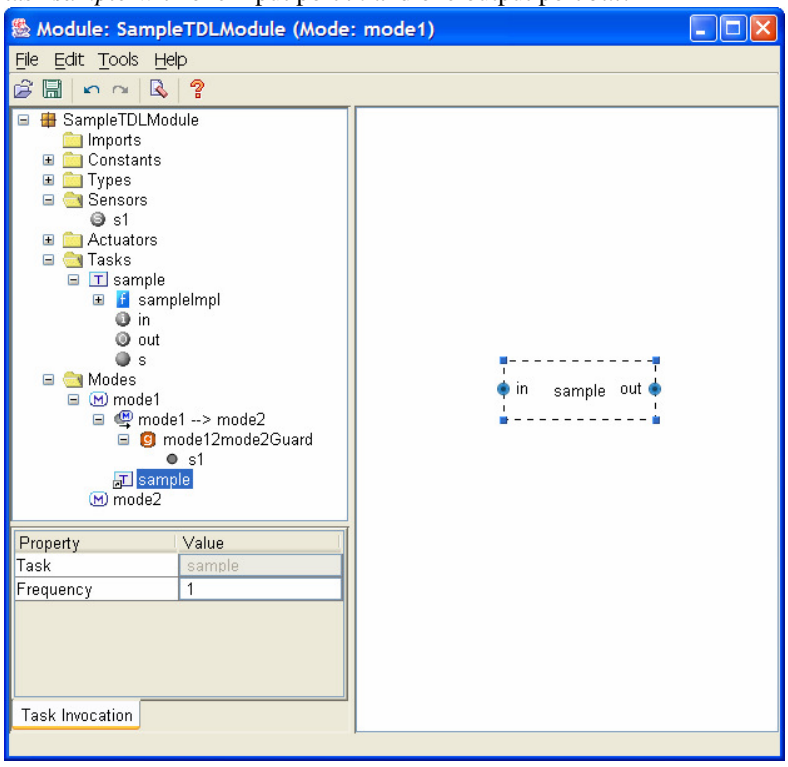
- You activate the Mode Transition editor again either by selecting the Modes folder in the tree, or by selecting the module name in the tree:



- The **Mode editor** allows the specification of task invocations and actuator updates. You activate the Mode editor by selecting a particular mode in the Modes folder. When you start editing a mode, the editor is empty such as in the following example:



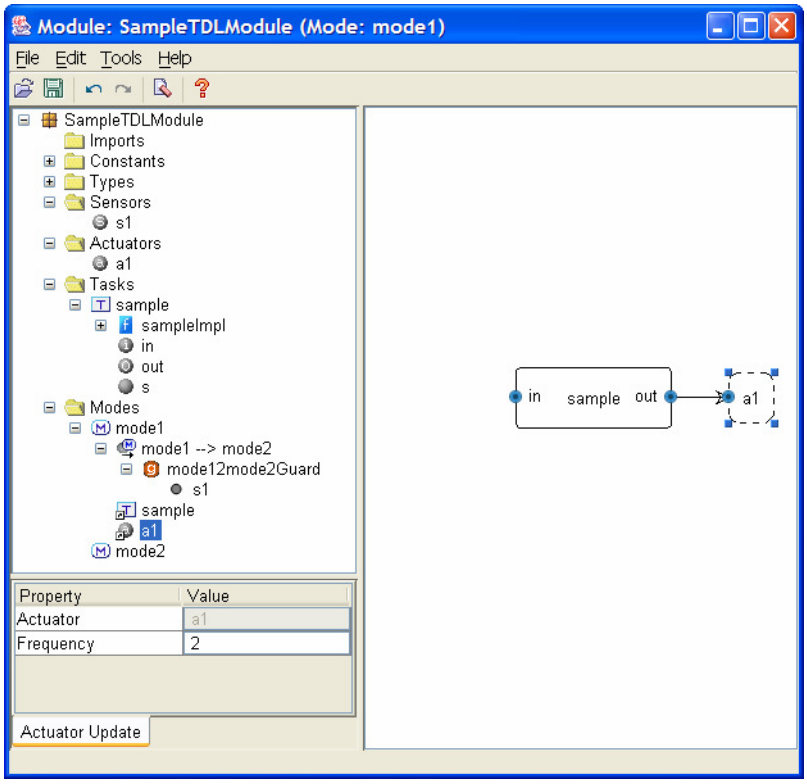
Task invocations. Select the task that should be invoked in the edited mode in the Tasks folder of the tree. Then drag it into the Mode editor. The Mode editor then displays the task as box with its name, such as task *sample* with one input port *in* and one output port *out*:



You change the task invocation Frequency in the inspector.
The LET of the task for this invocation can be specified explicitly by

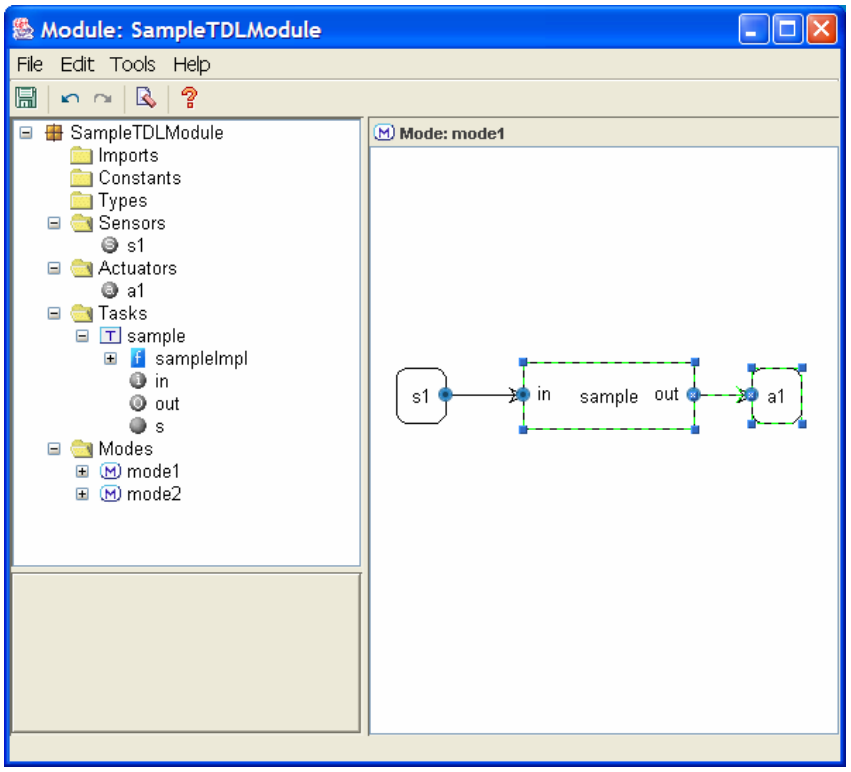
changing the Slots property (not shown in the screenshots). The default value '1*' executes the task with the specified frequency and with LET = mode period / frequency. For more information about slot selection we refer to the TDL Report in the meanwhile.

Actuator updates. In an analogous way as tasks you drag and drop actuators into the Mode editor and set the actuator update frequency. The following screen shot shows the Mode editor after dropping an actuator and setting the update frequency to 2:

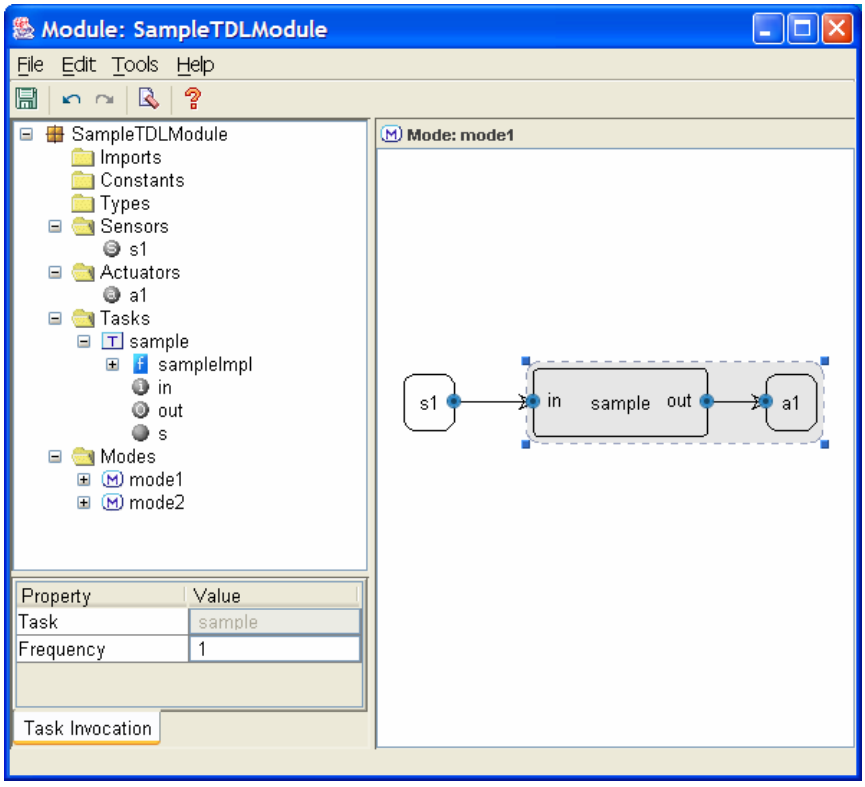


As sources for task input parameters you can use sensors (drag and drop them as described for tasks and actuators) or output ports of other tasks of the particular mode. Furthermore, the public sensor, actuator and task output ports of all imported modules are available.

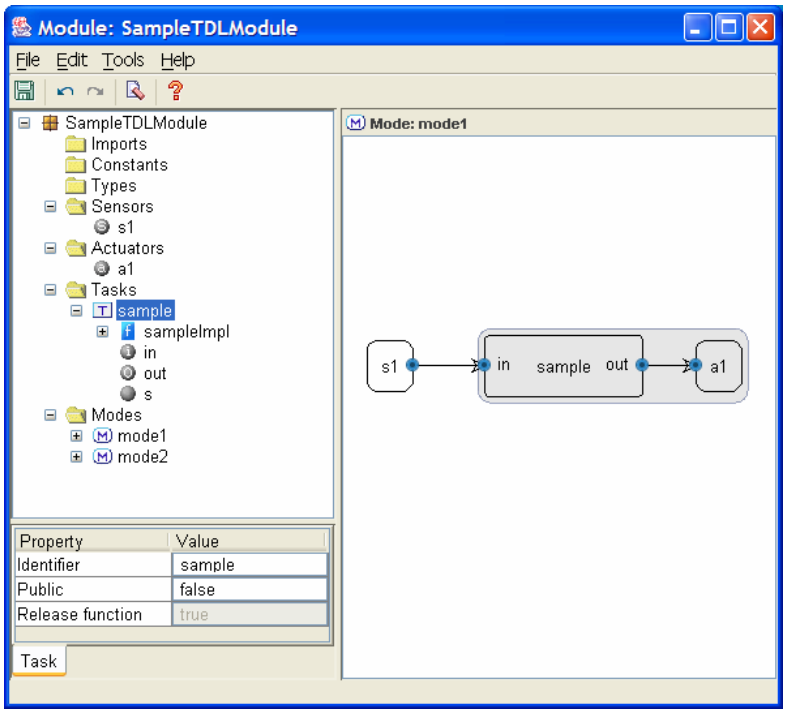
Actuator updates without LET delay. Digital controllers might require an actuator update as soon as possible, not after a task's LET period. To define that an actuator update should be accomplished synchronously (in locigal zero time after a task invocation) you select both the task declaration and the actuator in the Mode editor:



Then you right-click the mouse and choose ‘Group to sequence’ from the pop-up menu. The grouped task and actuator appear in a bounded gray box:



If you click then on the task in the tree, the property ‘Release function’ is set to true:

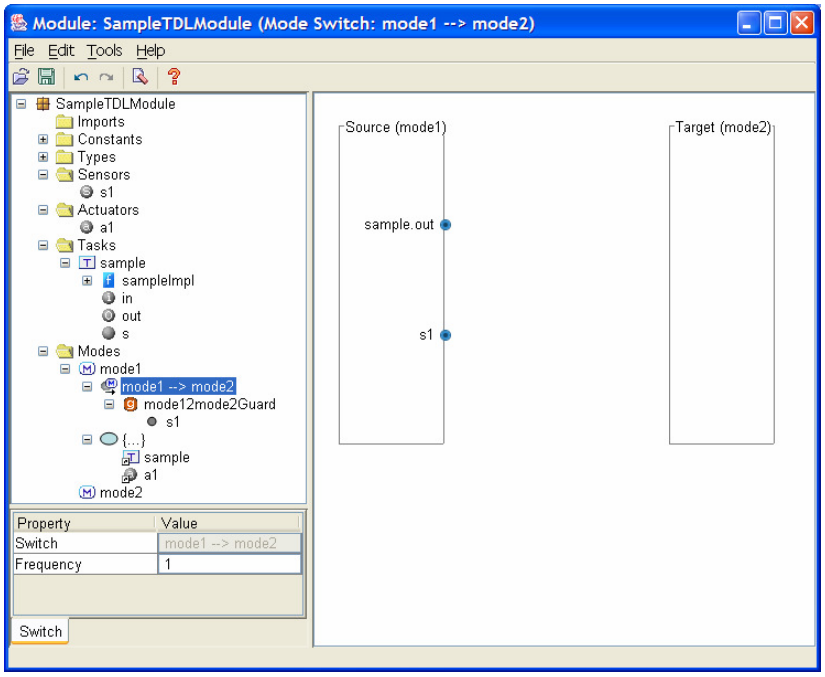


This means that the task execution is split in two functions, the release part and the one that can be used for advance calculations. For details we refer to the definition of the task sequence in the TDL language report. The TDL source code fragment illustrates which effect the definition of a task sequence has on the textual version of the task definition:

```
// ...
task sample {
    input
           double in;
    output
           double out;
    state
           double s;
    uses [release] sampleRelease(in,out);
    uses sampleImpl(in,out);
}

// ...
start mode mode1 [period=10 ms] {
     task
        [freq=1] {sample {in := s1;};
                  a1 := sample.out; }
    // ...
}
// ...
```

- **Port Assignment editor.** Assignments must be to output ports of tasks invoked in the target mode and must be thought of as initializations carried out as a first step in the affected target task's functionality code.

  If you need to assign values to output ports of tasks invoked in the target mode, for example, from the output values of tasks in the source mode, then you need to activate the Port Assignment editor. Click on the mode switch in the tree where you want to define the port assignments: You see the mode switches listed as sourceMode --> targetMode. For example, you activate the Port Assignment editor in our sample TDL module by clicking on *mode1-->mode2* in the tree:

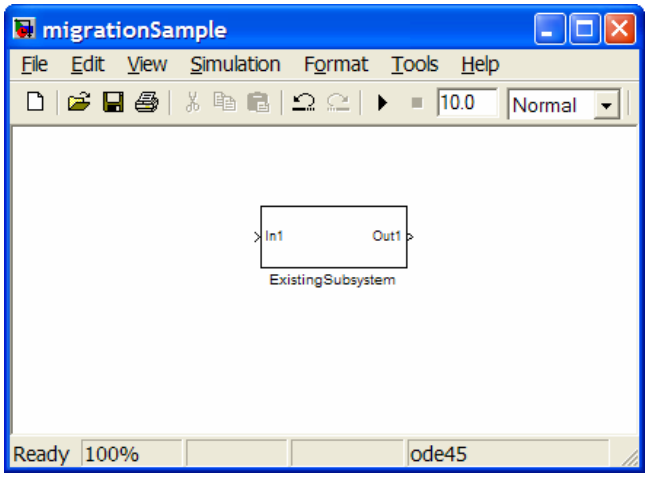


  The port assignment is defined by connecting the particular ports. In the example above this does not make sense as the target mode has no ports yet.

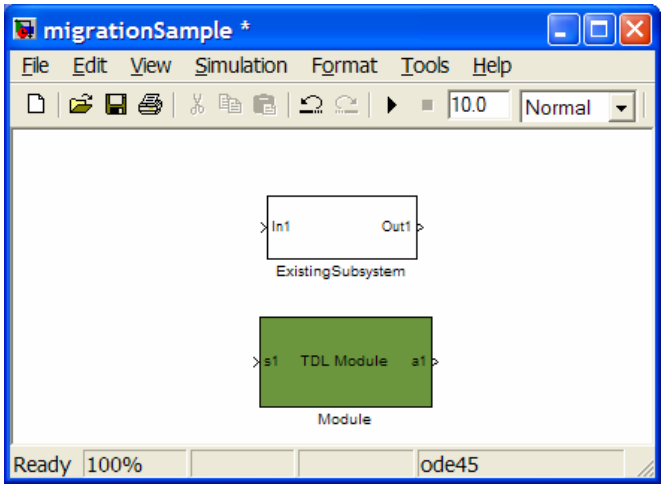
For further details we refer to the EBNF definition of this TDL construct.

Migration of existing Matlab/Simulink® subsystems to TDL modules

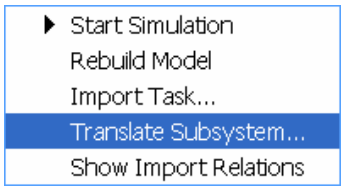
If you have already Matlab/Simulink® models of your controllers the TDL:VisualCreator supports an automated transfer of these models so that they become TDL modules. Let us consider a simple example. We assume that your controller model is represented in a subsystem called ExistingSubsystem:



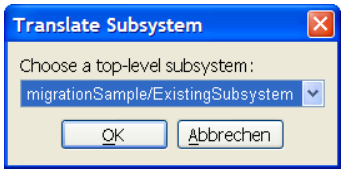
To transform the ExistingSubsystem so that it becomes a TDL module that executes this functionality as its only task, we copy a TDL module from the Simulink® library browser:



Double-click on the Module block to open the corresponding TDL:VisualCreator tool and choose 'Translate Subsystem...' from the Tools menu:

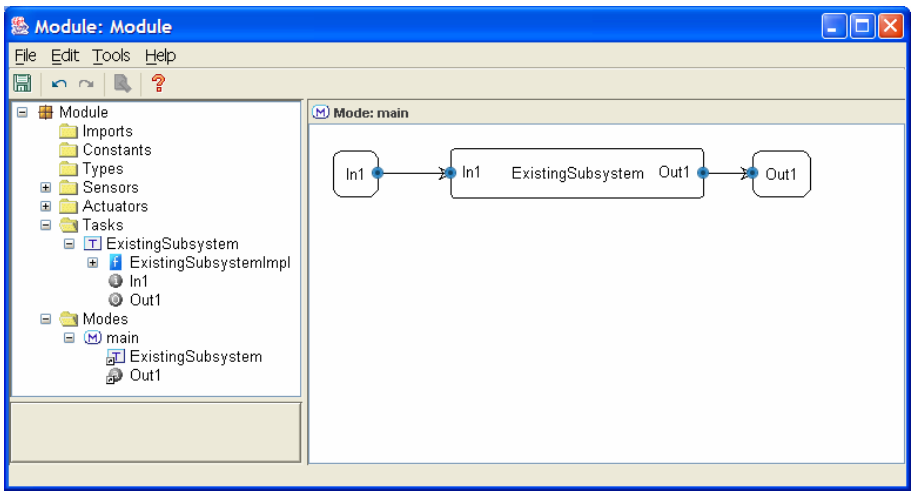


Then select the subsystem you want to translate in the dialog box and press the OK button:

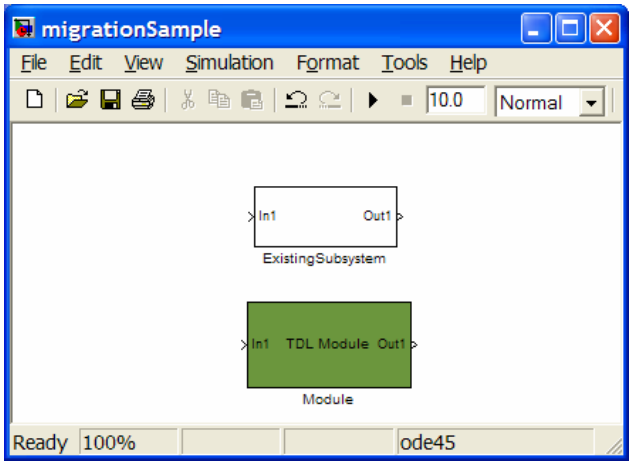


The result is that the selected subsystem becomes a task in the new module, with the following translations: the input ports of the subsystem become the sensors of the module, the output ports of the subsystem become the actuators of the module. A

task is created that has the same name as the original subsystem and that uses that subsystem as task function implementation. The corresponding input and output ports of the task are created automatically. The task is invoked within a mode (‘main’), where the sensors and actuators are automatically connected to the task ports.

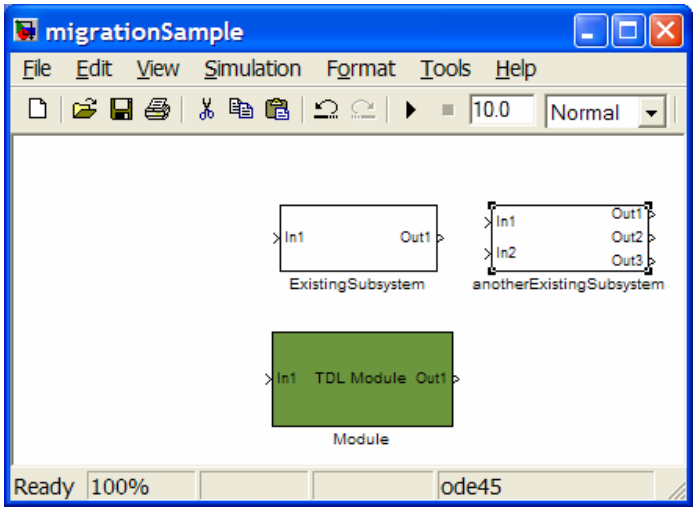


On the top level of the model, the TDL module has the same input ports (corresponding to the module’s sensors) and output ports (corresponding to the module’s actuators) as the original subsystem:

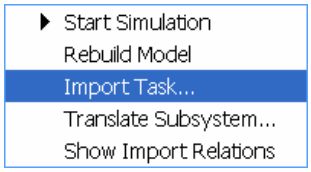


If necessary, adapt the mode period and, respectively, the frequency of the task invocation accordingly.

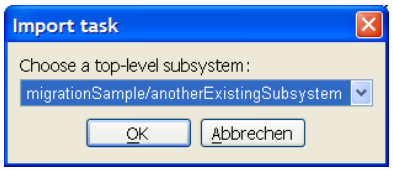
Adding a subsystem as task to a TDL module. The translation feature illustrated above is useful if you want one subsystem to become a TDL task in a TDL module. If you want several subsystems to become TDL tasks in one module, the TDL:VisualCreator offers an appropriate editing support either. Let us assume we want to add another existing subsystem as task to the TDL module we just have defined.



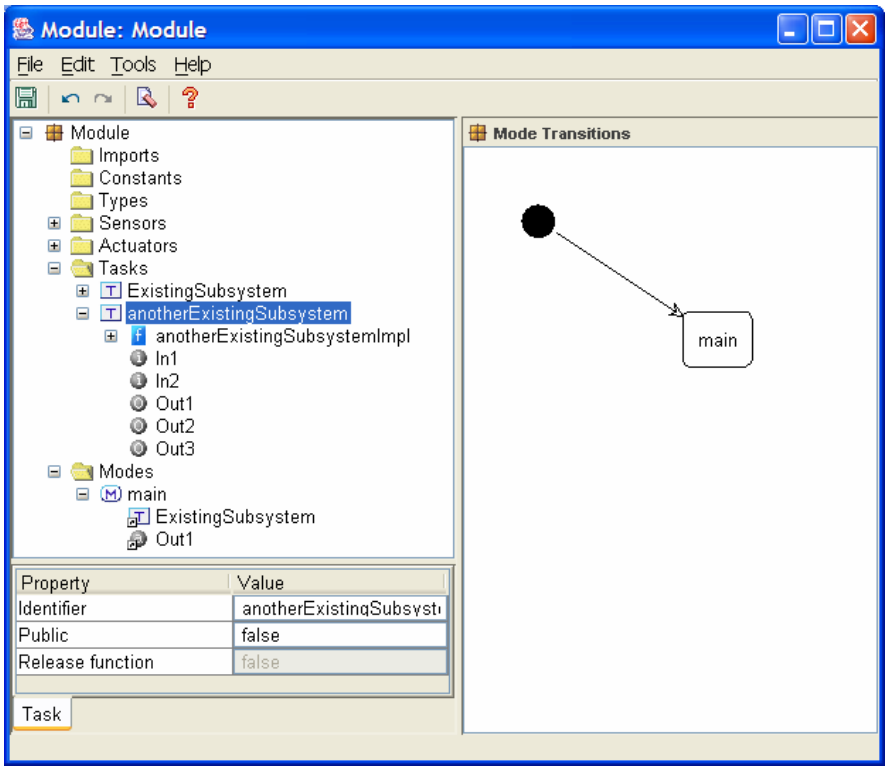
Then we open the TDL module with a double-click and choose the 'Import task...' item from the Tools menu.



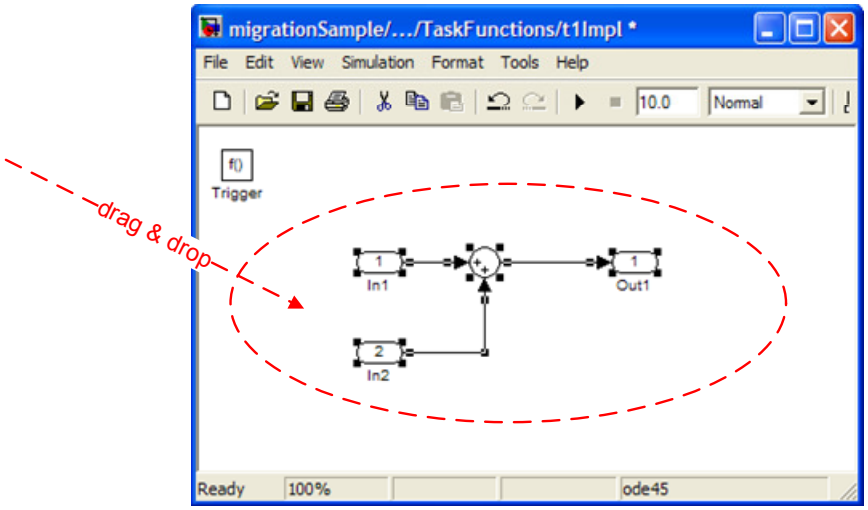
Then we select the subsystem we want to incorporate as task in the module.



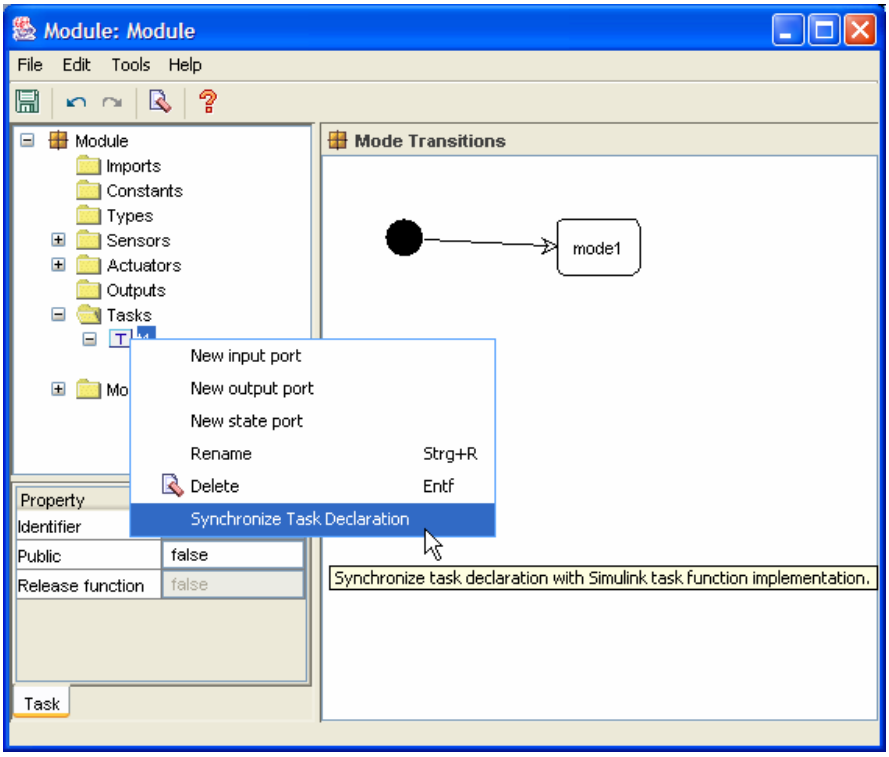
This results in an additional TDL task declaration with the input and output ports corresponding to the existing subsystem. Note that in this case no extra sensors and actuators are created. The developer also has to specify in which mode the task should be invoked.



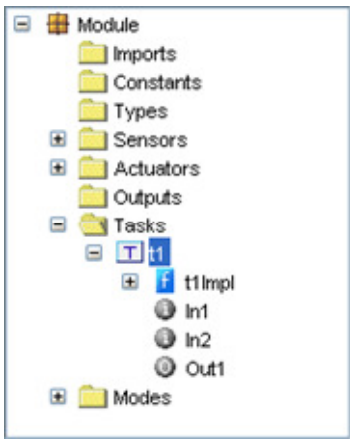
Copying models into TDL task functions. Another way of using existing subsystems as tasks is to copy the subsystem content into an already defined TDL task function. For that purpose we create a new TDL task in the TDL:VisualCreator tool and open its task function implementation by double-clicking on the function icon in the tree pane. We can now drag blocks from any open Simulink® model and drop them in our task function subsystem.

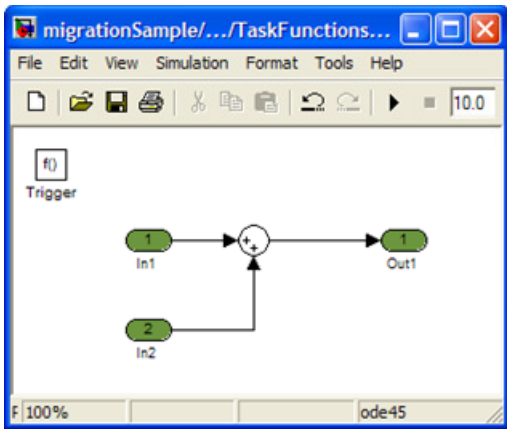


Subsequently, we ‘synchronize’ the TDL task declaration with our new task function, by right-clicking on the task function icon and choosing ‘Synchronize Task Declaration’ from the pop-up menu.



This will create a task input port for each Simulink® Inport and a task output port for each Outport in the task function.





Functionality code generation in Simulink®

When using the TDL:VisualCreator tool within Simulink® together with the Real-Time Workshop Embedded Coder®, functionality modelled in task-, guard-, and initializer-functions can be turned into C code.

Choosing the File>Export Functionality Code menu item starts the functionality code generation for the particular TDL module and places all files in a directory of your choice.

Please refer to the TDL:VisualDistributor tool documentation for more details about code generation particularly with regard to platform mapping and platform-specific code.

Simulink® Buses and TDL Structure Types

For each struct type t in module M, the TDL:VisualCreator tool automatically defines a Simulink.Bus object M_t in the Matlab® workspace. For code generation, a Simulink® BusCreator block that drives a TDL port block (i.e., sensor, actuator, or task port), must have its bus object property set to the corresponding Simulink.Bus object. Furthermore, input signals of a BusCreator block must be named according to their corresponding counterparts represented as struct elements of that type.

Appendix: The Timing Definition Language (TDL)

TDL is a pioneering domain-specific high-level modeling language for portable and deterministic hard real-time control applications. This chapter first explains TDL's advantages and argues why the state-of-the-art cannot deliver desired software properties such as determinism and portability. Finally, we describe all TDL language constructs.

TDL is in stark contrast to most state-of-that-art methodologies where the timing properties of software systems are accidental consequences of an implementation, rather than specified parts of the design. Concurrency often relies on low-level concepts such as semaphores. Further low-level concepts such as task priorities and interrupts with which developers have to cope with significantly increase the complexity of the software. The result is that when software is ported from one platform to another, timing changes significantly, often leading to significantly different computations and sometimes to malfunctioning systems.

Whereas with TDL (1) the functional and temporal behavior of a system is the same no matter where a TDL component is executed and (2) the developer does not have to care about the differences of local versus distributed execution of a TDL component. We guarantee that the software is free of deadlocks, race conditions and timing behavior that does not exactly correspond to the behavior specified in TDL. We refer to (1) and (2) as *transparent distribution*.

The TDL products have been built on the basis of cutting-edge research results in the realm of the Giotto project (http://www-cad.eecs.berkeley.edu/research/giotto/) at the University of California, Berkeley.

TDL delivers deterministic, portable software components

From a technical perspective the concept of Logical Execution Time (LET) is a key element in TDL. It allows us to deliver essential software properties that are necessary in order to efficiently produce ultra-reliable embedded software: **determinism, portability and composability**. In other words, these software properties imply **significant development, test and maintenance cost savings while improving the quality** of the embedded software. So TDL is a truly disruptive technology. You get correct software at a fraction of the costs you would have to invest with state-of-the-art methods and tools.

Determinism

If a software component is called twice with the same input values at the same time instants, it should both times produce the same output values at the same time instants.

Example: If a control component receives the same sensor values, it should always react exactly in the same way.

Consequences:

- Determinism is crucial in the context of product liability
- Testability: each behavior can be reproduced.
- Determinism is the precondition for ultra-reliable systems. Most of the currently available embedded software is not deterministic (see the explanation below).

Portability

The behavior of a software component should be specified independent of its implementation.

Example: The hardware, operating system, or bus architecture can be changed without changing the behavior of the application components.

Consequences:

- Protection of development and testing investments.
- Upgradability of hardware.

Composability

The behavior of a software component should be independent of the overall system load and configuration.

Example: A new component can be added to a system without changing the behavior of the original components.

Consequences:

- Significantly reduced costs for component integration.
- Extensibility and maintainability of systems.
- Reduced development costs through reuse of components.

The TDL tools are world-wide the first ones that can deliver all these properties of embedded software. The basis for that are the TDL module concept, invented in the realm of the MoDECS (Model-Based Development of Embedded Control Software) project (http://www.MoDECS.cc), and an adequate abstraction of the timing behavior (called the LET abstraction) from the underlying platform, invented at the University of California, Berkeley.

TDL concepts

TDL separates timing concerns, such as sampling rates and output jitter, from functionality concerns, such as control laws and device drivers. A TDL program supervises the interaction between software processes and the physical world, but does not itself transform data. All computation is encapsulated inside the supervised software processes, which can be written in any non-embedded programming language, such as C. We refer to a TDL program as a *timing program*, and to the

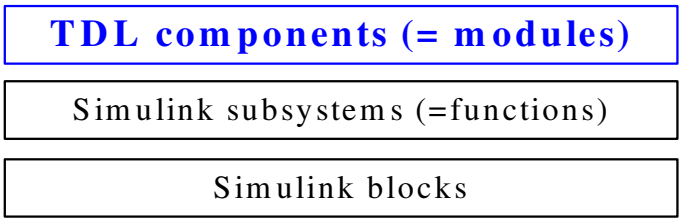
supervised processes called by the TDL program as *functionality programs*. The functionality programs can be modeled in Simulink. C source code can be generated from the Simulink models. Note that this separation of timing and functionality also protects intellectual property, that is, the control laws that correspond to the functionality programs.

In other words, TDL enforces a programming discipline—the strict separation of timing and functionality—which is often violated in real-time programs, especially when the programs are hand-optimized to meet timing constraints. This programming discipline offers additional benefits in the verification, optimization, integration, and reuse of embedded components. All of these activities are easier to carry out independently for timing and functionality programs. Similarly, the separation of timing and functionality further facilitates code generation: the TDL program can be compiled into executable timing code, independent of the compilation of the functionality (e.g., C) programs into executable pieces of functionality code. After linking, the timing code supervises the execution of the functionality code.

TDL component model

Can you imagine modern programming without classes, modules or components?

We assume that we use Matlab/Simulink® as modeling tool for control applications. Though Simulink® is a great tool for modeling and simulating controller functionality, it lacks the modeling constructs for building software components. Simulink's structuring unit is called a subsystem which corresponds to functions and procedures in programming languages. Modules or classes for defining components are missing. This is what TDL adds: a way to specify components and their interactions, in particular their temporal behavior.

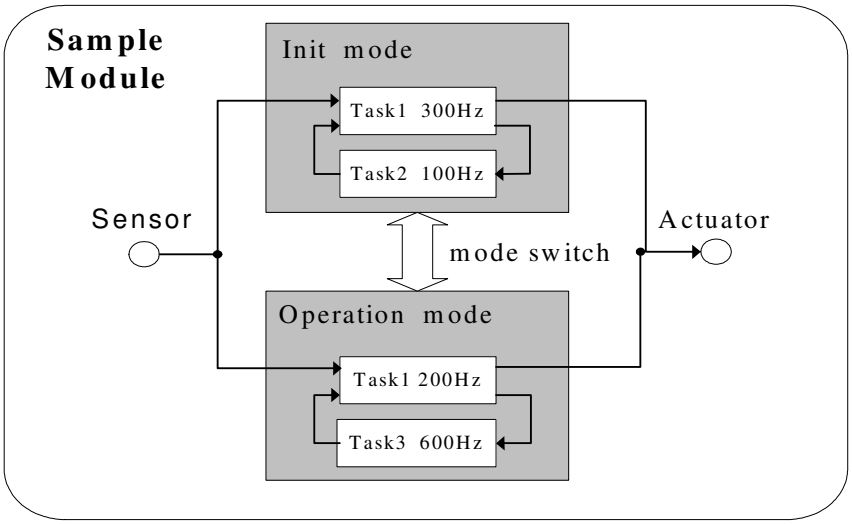


Due to the clear separation of timing and functionality in TDL, the functionality, that is the control laws, can be implemented in any language, for example, in C, generated from your Simulink models. In other words, with TDL components we have added the missing link for structuring hard real-time embedded software systems.

TDL modules serve multiple purposes. In addition to structuring your software, TDL modules are also the unit of distribution. So you can map TDL modules to nodes on your distributed platform. The code generators create the code that ensures that the modules behave exactly as specified in TDL. This is another significant advantage compared to Simulink®: The code generated for Simulink® subsystems cannot easily be mapped to computing nodes in a distributed system.

The following figure illustrates schematically a TDL component (= module): a TDL module forms a unit that consists of sensors, actuators, and modes. A mode is a set of periodically executed activities. The activities are task invocations, actuator updates, and mode switches. All activities can have their own rate of execution and all activities can be executed conditionally. The tasks represent the functionality, particularly the control laws. As mentioned above any suitable language can be

chosen to implement the functionality. If MathWorks' tools are used, the functionality is modeled in Simulink®.

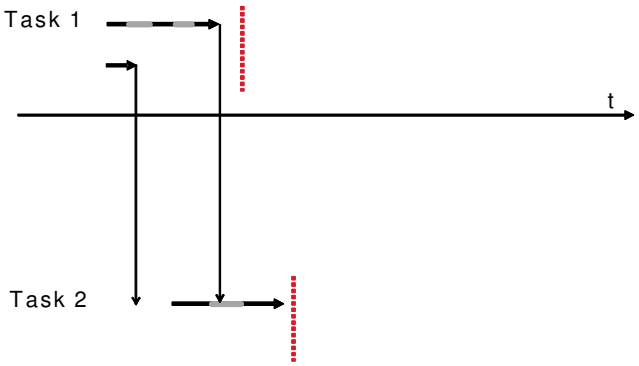


Actuator updates and mode switches are considered to be much faster than task invocations, thus they are executed in logical zero time. Whereas task invocations adhere to the so called Logical Execution Time (LET) semantics.

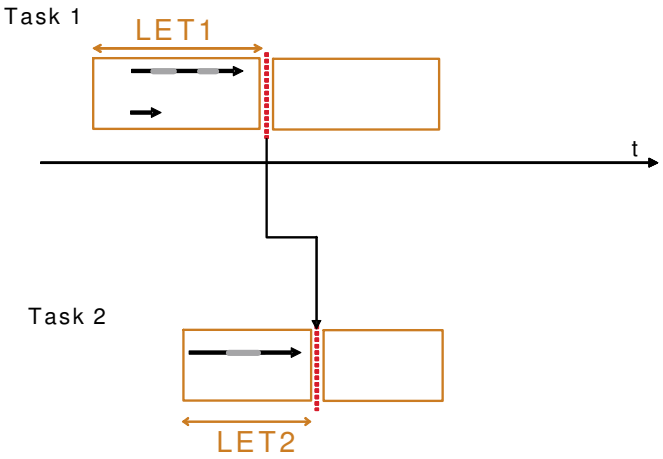
Comparison with the state-of-the-art

Why is state-of-the-art embedded software not deterministic, not portable, not composable?

Embedded software for safety-critical applications typically does the following periodically: The inputs are read from sensors (for example, the rotation speed of the wheels of a vehicle), processed by a function and the results are forwarded to other functions or actuators as soon as the calculation is finished. The only constraint is that the function calculation finishes before a deadline. The problem is that this programming model leads to non-deterministic behavior as illustrated in the following figure. In the example the deadline for the two functions is shown as dotted red line. Thus the slower computation of function 1, indicated by the length of the upper arrow, is as fine as the faster computation, indicated by the short lower arrow, because both computations finish before the deadline of function 1. (The gray sections in the arrows indicate suspension times of the particular function.) Unfortunately, function 2, which requires the input from function 1, receives a different input, depending when function 1 finishes. This means that the behavior of the embedded system is not deterministic. If a faster processor is used, the behavior of the overall system might be different from the behavior produced by a slower processor. In other words, the embedded software depends on the platform. Each change might trigger unforseeable effects that might even cause catastrophic system behavior.



The LET abstraction is the basis for overcoming the mentioned problem. The following figure illustrates the LET abstraction in action: no matter how long the computation actually takes, the results are only delivered exactly after the LET of a function, in the example set exactly at the deadline. Because LET1 of Task 1's invocation finishes after the corresponding LET2 time period, the results are always available as input of the next invocation of Task 2.



LET has established a new programming paradigm for embedded software. This is comparable to the invention of the abstractions in higher programming languages such as Algol, Pascal, C, and later on C, C++, Java and C#. LET was invented in the realm of the Giotto project. Thus, Giotto has paved the way for higher level programming languages for embedded software. It focuses on the timing behavior. The implications and its benefits are manifold:

The LET abstraction allows the automatic software generation from a specification of the timing behavior, supposed that the particular platform is powerful enough to provide the required computing and communication resources. A platform change simply means a fully automated re-generation of the software.

For several reasons, such as fault-tolerance and scalability, safety-critical embedded systems are based on distributed computing platforms. The LET abstraction greatly simplifies the development of distributed software—the TDL tools automatically generate efficient distributed software from the LET-based timing behavior specification. The developer only has to map TDL modules with a simple drag&drop operation to the nodes of a distributed system. FlexRay is a sample target platform for which the code generators have already been implemented and used.

The other aspects, in particular, the implementation of the functions, can be accomplished in any programming language or with any tool, such as MathWorks' tool suite.

Note that the TDL tools form a natural synergy with existing embedded software development tools that focus on functionality, in particular with MathWorks' tool suite. Thus, the TDL tools do not replace them, but complement them.